IMSL®

IMSL Fortran Library User's Guide MATH/LIBRARY Volume 2 of 2

Mathematical Functions in Fortran

Trusted For Over 30 Years

IMSL®

IMSL Fortran Library User's Guide
MATH/LIBRARY Volume 2 of 2

Mathematical Functions in Fortran

Visual Numerics

[www.vni.com]

P/N 7694

Visual Numerics, Inc. – United States
Corporate Headquarters
2000 Crow Canyon Place, Suite 270
San Ramon, CA 94583
PHONE: 925-807-0138
FAX: 925-807-0145
e-mail: info@vni.com
Westminster, CO
PHONE: 303-379-3040

Houston, TX
PHONE: 713-784-3131

Visual Numerics International Ltd.
Sussex House
6 The Forbury
Reading, Berkshire RGI 3EJ
UNITED KINGDOM

PHONE: +44-1-189 25-3370
FAX: +44 –1-189-25-3371
e-mail: info@vniuk.co.uk
Support: support@vniuk.co.uk

Visual Numerics SARL
Immeuble le Wilson 1
70, avenue due General de Gaulle
F-92058 PARIS LA DEFENSE, Cedex
FRANCE

PHONE: +33-1-46-93-94-20
FAX: +33-1-46-93-94-39
e-mail: info@vni.paris.fr

Visual Numerics S. A. de C. V.
Florencia 57 Piso 10-01
Col. Juarez
Mexico D. F. C. P. 06600
MEXICO

PHONE: +52-55-514-9730 or 9628
FAX: +52-55-514-4873

Visual Numerics International GmbH
Zettachring 10
D-70567Stuttgart
GERMANY

PHONE: +49-711-13287-0
FAX: +49-711-13287-99
e-mail: vni@visual-numerics.de

Visual Numerics Japan, Inc.
GOBANCHO HIKARI BLDG. 4TH Floor
14 GOBAN-CHO CHIYODA-KU
TOKYO, JAPAN 102

PHONE: +81-3-5211-7760
FAX: +81-3-5211-7769
e-mail: vnijapan@vnij.co.jp

Visual Numerics, Inc.
7/F, #510, Sect. 5
Chung Hsiao E. Road
Taipei, Taiwan 110
ROC

PHONE: +(886) 2-2727-2255
FAX: +(886) 2-2727-6798
e-mail: info@vni.com.tw

Visual Numerics Korea, Inc.
HANSHIN BLDG. Room 801
136-1, MAPO-DONG, MAPO-GU
SEOUL, 121-050
KOREA SOUTH

PHONE: +82-2-3273-2632 or 2633
FAX: +82-2-3273--2634
e-mail: info@vni.co.kr

World Wide Web site: http://www.vni.com

The information contained in this document is subject to change without notice.

This document is provided AS IS, with NO WARRANTY. VISUAL NUMERICS, INC., SHALL NOT BE LIABLE FOR ANY ERRORS WHICH MAY BE CONTAINED HEREIN OR FOR INCIDENTAL, CONSEQUENTIAL, OR OTHER INDIRECT DAMAGES IN CONNECTION WITH THE FURNISHING, PERFORMANCE OR USE OF THIS MATERIAL. [Carol: note case change]

IMSL Fortran, C, and Java
Application Development Tools

Contents

Volume I

Volume II

Chapter 5: Differential Equations

Routines

Usage Notes

A *differential equation* is an equation involving one or more dependent variables (called y_i or u_i), their derivatives, and one or more independent variables (called t, x, and y). Users will typically need to relabel their own model variables so that they correspond to the variables used in the solvers described here. A differential equation with one independent variable is called an *ordinary differential equation* (ODE). A system of equations involving derivatives in one independent variable and other dependent variables is called a *differential-algebraic system*. A differential equation with more than one independent variable is called a *partial differential equation* (PDE). The *order* of a differential equation is the highest order of any of the derivatives in the equation. Some of the routines in this chapter require the user to reduce higher-order problems to systems of first-order differential equations.

Ordinary Differential Equations

It is convenient to use the vector notation below. We denote the number of equations as the value N. The problem statement is abbreviated by writing it as a *system* of first-order ODEs

$$y(t) = \left[y_1(t), \ldots, y_N(t)\right]^T, f(t, y) = \left[f_1(t, y), \ldots, f_N(t, y)\right]^T$$

The problem becomes

$$y' = \frac{dy(t)}{dt} = f(t, y)$$

with initial values $y(t_0)$. Values of $y(t)$ for $t > t_0$ or $t < t_0$ are required. The routines `IVPRK`, page 837, `IVMRK`, page 844, and `IVPAG`, page 854, solve the IVP for systems of ODEs of the form $y' = f(t, y)$ with $y(t = t_0)$ specified. Here, f is a user supplied function that must be evaluated at any set of values $(t, y_1, \ldots, y_N)$; $i = 1, \ldots, N$. The routines `IVPAG`, page 854, and `DASPG`, page 889, will also solve implicit systems of the form $Ay' = f(t, y)$ where A is a user supplied matrix. For `IVPAG`, the matrix A must be nonsingular.

The system $y' = f(t, y)$ is said to be *stiff* if some of the eigenvalues of the Jacobian matrix $\{\partial f_i/\partial y_j\}$ have large, negative real parts. This is often the case for differential equations representing the behavior of physical systems such as chemical reactions proceeding to equilibrium where subspecies effectively complete their reaction in different epochs. An alternate model concerns discharging capacitors such that different parts of the system have widely varying decay rates (or *time constants*). This definition of stiffness, based on the eigenvalues of the Jacobian matrix, is not satisfactory. Users typically identify stiff systems by the fact that numerical differential equation solvers such as `IVPRK`, page 837, are inefficient, or else they fail. The most common inefficiency is that a large number of evaluations of the functions f_i are required. In such cases, use routine `IVPAG`, page 854, or `DASPG`, page 889. For more about stiff systems, see Gear (1971, Chapter 11) or Shampine and Gear (1979).

In the *boundary value problem* (BVP) for ODEs, constraints on the dependent variables are given at the endpoints of the interval of interest, $[a, b]$. The routines `BVPFD`, page 889, and `BVPMS`, page 882, solve the BVP for systems of the form $y'(t) = f(t, y)$, subject to the conditions

$$h_i(y_1(a), \ldots, y_N(a), y_1(b), \ldots, y_N(b)) = 0 \quad i = 1, \ldots, N$$

Here, f and $h = [h_1, \ldots, h_N]^T$ are user-supplied functions.

Differential-algebraic Equations

Frequently, it is not possible or not convenient to express the model of a dynamical system as a set of ODEs. Rather, an implicit equation is available in the form

$$g_i\left(t, y, \ldots, y_N, y_1', \ldots, y_N'\right) = 0 \qquad i = 1, \ldots, N$$

The g_i are user-supplied functions. The system is abbreviated as

$$g\left(t, y, y'\right) = \left[g_1\left(t, y, y'\right), \ldots, g_N\left(t, y, y'\right)\right]^T = 0$$

With initial value $y(t_0)$. Any system of ODEs can be trivially written as a differential-algebraic system by defining

$$g\left(t, y, y'\right) = f\left(t, y\right) - y'$$

The routine `DASPG`, page 889, solves differential-algebraic systems of index 1 or index 0. For a definition of *index* of a differential-algebraic system, see (Brenan et al. 1989). Also, see Gear and Petzold (1984) for an outline of the computing methods used.

Partial Differential Equations

The routine `MOLCH`, page 946, solves the IVP problem for systems of the form

$$\frac{\partial u_i}{\partial t} = f_i\left(x, t, u_1, \ldots, u_N, \frac{\partial u_1}{\partial x}, \ldots, \frac{\partial u_N}{\partial x}, \frac{\partial^2 u_1}{\partial x^2}, \ldots, \frac{\partial^2 u_N}{\partial x^2}\right)$$

subject to the boundary conditions

$$\begin{aligned} \alpha_1^{(i)} u_i\left(a\right) + \beta_1^{(i)} \frac{\partial u_i}{\partial x}\left(a\right) &= \gamma_1\left(t\right) \\ \alpha_2^{(i)} u_i\left(b\right) + \beta_2^{(i)} \frac{\partial u_i}{\partial x}\left(b\right) &= \gamma_2\left(t\right) \end{aligned}$$

and subject to the initial conditions

$$u_i(x, t = t_0) = g_i(x)$$

for $i = 1, \ldots, N$. Here, f_i, g_i,

$$\alpha_j^{(i)}, \text{ and } \beta_j^{(i)}$$

are user-supplied, $j = 1, 2$.

The routines `FPS2H`, page 961, and `FPS3H`, page 967, solve Laplace's, Poisson's, or Helmholtz's equation in two or three dimensions. `FPS2H` uses a fast Poisson method to solve a PDE of the form

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + cu = f\left(x, y\right)$$

over a rectangle, subject to boundary conditions on each of the four sides. The scalar constant c and the function f are user specified. `FPS3H` solves the three-dimensional analogue of this problem.

Users wishing to solve more general PDE's, in more general 2-d and 3-d regions are referred to Visual Numerics' partner PDE2D (www.pde2d.com).

Summary

The following table summarizes the types of problems handled by the routines in this chapter. With the exception of `FPS2H` and `FPS3H`, the routines can handle more than one differential equation.

Problem	Consideration	Routine
$Ay' = f(t, y)$ $y(t_0) = y_0$	A is a general, symmetric positive definite, band or symmetric positive definite band matrix.	`IVPAG` page 854
	Stiff or expensive to evaluate $f(t, y)$, banded Jacobian or finely spaced output needed.	`IVPAG` page 854
$y' = f(t, y)$, $y(t_0) = y_0$	High accuracy needed and not stiff. (Uses Adams methods)	`IVPAG` page 854
	Moderate accuracy needed and not stiff.	`IVPRK` page 837
$y' = f(t, y)$ $h(y(a), y(b)) = 0$	BVP solver using finite differences	`BVPFD` page 870
	BVP solver using multiple shooting	`BVPMS` page 882
$g(t, y, y') = 0$ $y(t_0)$, $y'(t_0)$ given	Stiff, differential-algebraic solver for systems of index 1 or 0. **Note:** `DASPG` uses the user-supplied $y'(t_0)$ only as an initial guess to help it find the correct initial $y'(t_0)$ to get started.	`DASPG` page 889
$u_t = f(x, t, u, u_x, u_{xx})$ $\alpha_1 u(a) + \beta_1 u_x(a) = \gamma_1(t)$ $\alpha_2 u(b) + \beta_2 u_x(b) = \gamma_2(t)$	Method of lines using cubic splines and ODEs.	`MOLCH` page 946
$u_{xx} + u_{yy} + cu = f(x, y)$ on a rectangle, given u or u_n on each edge.	Fast Poisson solver	`FPS2H` page 961
$u_{xx} + u_{yy} + u_{zz} + cu = f(x, y, z)$ on a box, given u or u_n on each face	Fast Poisson solver	`FPS3H` page 967
$-(pu')' + qu = \lambda ru,$ $\alpha_1 u(a) - \alpha_2 (pu'(a))$ $= \lambda(\alpha_1' u(a) - \alpha_2' (pu'(a)))$ $\beta_1 u(b) + \beta_2 (pu'(b)) = 0$	Sturm-Liouville problems	`SLEIG` page 973

IVPRK

Solves an initial-value problem for ordinary differential equations using the Runge-Kutta-Verner fifth-order and sixth-order method.

Required Arguments

IDO — Flag indicating the state of the computation. (Input/Output)

`IDO`	**State**
1	Initial entry
2	Normal re-entry
3	Final call to release workspace
4	Return because of interrupt 1
5	Return because of interrupt 2 with step accepted
6	Return because of interrupt 2 with step rejected

Normally, the initial call is made with `IDO` = 1. The routine then sets `IDO` = 2, and this value is used for all but the last call that is made with `IDO` = 3. This final call is used to release workspace, which was automatically allocated by the initial call with `IDO` = 1. No integration is performed on this final call. See Comment 3 for a description of the other interrupts.

FCN — User-supplied `SUBROUTINE` to evaluate functions. The usage is `CALL FCN(N, T, Y, YPRIME)`, where

`N` – Number of equations. (Input)
`T` – Independent variable, *t*. (Input)
`Y` – Array of size `N` containing the dependent variable values, *y*. (Input)
`YPRIME` – Array of size `N` containing the values of the vector *y′* evaluated at (*t,* y). (Output)

`FCN` must be declared `EXTERNAL` in the calling program.

T — Independent variable. (Input/Output)
On input, `T` contains the initial value. On output, `T` is replaced by `TEND` unless error conditions have occurred. See `IDO` for details.

TEND — Value of *t* where the solution is required. (Input)
The value `TEND` may be less than the initial value of *t*.

Y — Array of size NEQ of dependent variables. (Input/Output)
On input, Y contains the initial values. On output, Y contains the approximate solution.

Optional Arguments

NEQ — Number of differential equations. (Input)
Default: NEQ = size (Y,1).

TOL — Tolerance for error control. (Input)
An attempt is made to control the norm of the local error such that the global error is proportional to TOL.
Default: TOL = machine precision.

PARAM — A *floating-point* array of size 50 containing optional parameters. (Input/ Output)
If a parameter is zero, then a default value is used. These default values are given below. Parameters that concern values of step size are applied in the direction of integration. The following parameters may be set by the user:

	PARAM	**Meaning**
1	HINIT	Initial value of the step size. Default: 10.0 * MAX (AMACH (1), AMACH(4) * MAX(ABS(TEND), ABS(T)))
2	HMIN	Minimum value of the step size. Default: 0.0
3	HMAX	Maximum value of the step size. Default: 2.0
4	MXSTEP	Maximum number of steps allowed. Default: 500
5	MXFCN	Maximum number of function evaluations allowed. Default: No enforced limit.
6		Not used.
7	INTRP1	If nonzero, then return with IDO = 4 before each step. See Comment 3. Default: 0.
8	INTRP2	If nonzero, then return with IDO = 5 after every successful step and with IDO = 6 after every unsuccessful step. See Comment 3. Default: 0.
9	SCALE	A measure of the scale of the problem, such as an approximation to the average value of a norm of the Jacobian matrix along the solution. Default: 1.0

	PARAM	Meaning
10	INORM	Switch determining error norm. In the following, e_i is the absolute value of an estimate of the error in $y_i(t)$. Default: 0.0 – min(absolute error, relative error) = max(e_i/w_i); $i = 1, \ldots, NEQ$, where $w_i = \max(\lvert y_i(t)\rvert, 1.0)$. 1 – absolute error = max(e_i), $i = 1 \ldots, NEQ$. 2– max(e_i/w_i), $i = 1 \ldots, NEQ$ where $w_i = \max(\lvert y_i\ (t)\rvert$, FLOOR), and FLOOR is PARAM(11). 3 – Scaled Euclidean norm defined as where $w_i = \max(\lvert y_i\ (t)\rvert, 1.0)$. Other definitions of YMAX can be specified by the user, as explained in Comment 1.
11	FLOOR	Used in the norm computation associated with parameter INORM. Default: 1.0.
12–30		Not used.

The following entries in PARAM are set by the program.

	PARAM	Meaning
31	HTRIAL	Current trial step size.
32	HMINC	Computed minimum step size allowed.
33	HMAXC	Computed maximum step size allowed.
34	NSTEP	Number of steps taken.
35	NFCN	Number of function evaluations used.
36–50		Not used.

FORTRAN 90 Interface

Generic: `CALL IVPRK (IDO, FCN, T, TEND, Y [,…])`

Specific: The specific interface names are `S_IVPRK` and `D_IVPRK`.

FORTRAN 77 Interface

Single: `CALL IVPRK (IDO, NEQ, FCN, T, TEND, TOL, PARAM, Y)`

Double: The double precision name is `DIVPRK`.

Example 1

Consider a predator-prey problem with rabbits and foxes. Let r be the density of rabbits and let f be the density of foxes. In the absence of any predator-prey interaction, the rabbits would increase at a rate proportional to their number, and the foxes would die of starvation at a rate proportional to their number. Mathematically,

$$r' = 2r$$

$$f' = -f$$

The rate at which the rabbits are eaten by the foxes is $2rf$, and the rate at which the foxes increase, because they are eating the rabbits, is rf. So, the model to be solved is

$$r' = 2r - 2rf$$

$$f' = -f + rf$$

The initial conditions are $r(0) = 1$ and $f(0) = 3$ over the interval $0 \le t \le 10$.

In the program `Y`(1) = r and `Y`(2) = f. Note that the parameter vector `PARAM` is first set to zero with IMSL routine `SSET` (Chapter 9, Basic Matrix/Vector Operations). Then, absolute error control is selected by setting `PARAM`(10) = 1.0.

The last call to `IVPRK` with `IDO` = 3 deallocates IMSL workspace allocated on the first call to `IVPRK`. It is not necessary to release the workspace in this example because the program ends after solving a single problem. The call to release workspace is made as a model of what would be needed if the program included further calls to IMSL routines.

```
      USE IVPRK_INT
      USE UMACH_INT
      INTEGER    MXPARM, N
      PARAMETER  (MXPARM=50, N=2)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    IDO, ISTEP, NOUT
      REAL       PARAM(MXPARM), T, TEND, TOL, Y(N)
!                                 SPECIFICATIONS FOR SUBROUTINES
      EXTERNAL   FCN
!
      CALL UMACH (2, NOUT)
!                                 Set initial conditions
      T = 0.0
      Y(1) = 1.0
      Y(2) = 3.0
!                                 Set error tolerance
      TOL = 0.0005
!                                 Set PARAM to default
      PARAM = 0.E0
!                                 Select absolute error control
      PARAM(10) = 1.0
!                                 Print header
      WRITE (NOUT,99999)
      IDO = 1
      ISTEP = 0
   10 CONTINUE
      ISTEP = ISTEP + 1
      TEND = ISTEP
      CALL IVPRK (IDO, FCN, T, TEND, Y, TOL=TOL, PARAM=PARAM)
      IF (ISTEP .LE. 10) THEN
         WRITE (NOUT,’(I6,3F12.3)’) ISTEP, T, Y
!                                 Final call to release workspace
```

```
         IF (ISTEP .EQ. 10) IDO = 3
         GO TO 10
      END IF
99999 FORMAT (4X, 'ISTEP', 5X, 'Time', 9X, 'Y1', 11X, 'Y2')
      END
      SUBROUTINE FCN (N, T, Y, YPRIME)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    N
      REAL       T, Y(N), YPRIME(N)
!
      YPRIME(1) = 2.0*Y(1) - 2.0*Y(1)*Y(2)
      YPRIME(2) = -Y(2) + Y(1)*Y(2)
      RETURN
      END
```

Output

```
 ISTEP     Time          Y1          Y2
 1       1.000       0.078       1.465
 2       2.000       0.085       0.578
 3       3.000       0.292       0.250
 4       4.000       1.449       0.187
 5       5.000       4.046       1.444
 6       6.000       0.176       2.256
 7       7.000       0.066       0.908
 8       8.000       0.148       0.367
 9       9.000       0.655       0.188
10      10.000       3.157       0.352
```

Comments

1. Workspace may be explicitly provided, if desired, by use of I2PRK/DI2PRK. The reference is:

   ```
   CALL I2PRK (IDO, NEQ, FCN, T, TEND, TOL, PARAM, Y,
   VNORM, WK)
   ```

   The additional arguments are as follows: YMAX $= \sqrt{\sum_{i=1}^{NEQ} e_i^2 / w_i^2}$

VNORM — A Fortran SUBROUTINE to compute the norm of the error. (Input)
The routine may be provided by the user, or the IMSL routine I3PRK/DI3PRK may be used. In either case, the name must be declared in a Fortran EXTERNAL statement. If usage of the IMSL routine is intended, then the name I3PRK/DI3PRK should be used. The usage of the error norm routine is CALL VNORM (N, V, Y, YMAX, ENORM), where

Arg	Definition
N	Number of equations. (Input)
V	Array of size N containing the vector whose norm is to be computed. (Input)

Y Array of size N containing the values of the dependent variable. (Input)

YMAX Array of size N containing the maximum values of $|y(t)|$. (Input)

ENORM Norm of the vector V. (Output)

VNORM must be declared EXTERNAL in the calling program.

WK — Work array of size 10N using the working precision. The contents of WK must not be changed from the first call with IDO = 1 until after the final call with IDO = 3.

2. Informational errors

Type	Code	
4	1	Cannot satisfy error condition. The value of TOL may be too small.
4	2	Too many function evaluations needed.
4	3	Too many steps needed. The problem may be stiff.

3. If PARAM(7) is nonzero, the subroutine returns with IDO = 4 and will resume calculation at the point of interruption if re-entered with IDO = 4. If PARAM(8) is nonzero, the subroutine will interrupt the calculations immediately after it decides whether or not to accept the result of the most recent trial step. The values used are IDO = 5 if the routine plans to accept, or IDO = 6 if it plans to reject the step. The values of IDO may be changed by the user (by changing IDO from 6 to 5) in order to force acceptance of a step that would otherwise be rejected. Some parameters the user might want to examine after return from an interrupt are IDO, HTRIAL, NSTEP, NFCN, T, and Y. The array Y contains the newly computed trial value for $y(t)$, accepted or not.

Description

Routine IVPRK finds an approximation to the solution of a system of first-order differential equations of the form $y_0 = f(t, y)$ with given initial data. The routine attempts to keep the global error proportional to a user-specified tolerance. This routine is efficient for nonstiff systems where the derivative evaluations are not expensive.

The routine IVPRK is based on a code designed by Hull, Enright and Jackson (1976, 1977). It uses Runge-Kutta formulas of order five and six developed by J. H. Verner.

Additional Examples

Example 2

This is a mildly stiff problem (F2) from the test set of Enright and Pryce (1987). It is included here because it illustrates the inefficiency of requiring more function evaluations with a nonstiff solver, for a requested accuracy, than would be required using a stiff solver. Also, see IVPAG, page 854, Example 2, where the problem is solved using a BDF method. The number of function evaluations may vary, depending on the accuracy and other arithmetic characteristics of the computer. The test problem has $n = 2$ equations:

$$
\begin{aligned}
y_1' &= -y_1 - y_1 y_2 + k_1 y_2 \\
y_2' &= -k_2 y_2 + k_3 \left(1 - y_2\right) y_1 \\
y_1\left(0\right) &= 1 \\
y_2\left(0\right) &= 0 \\
k_1 &= 294 \\
k_2 &= 3 \\
k_3 &= 0.01020408 \\
tend &= 240
\end{aligned}
$$

```
      USE IVPRK_INT
      USE UMACH_INT
      INTEGER    MXPARM, N
      PARAMETER  (MXPARM=50, N=2)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    IDO, ISTEP, NOUT
      REAL       PARAM(MXPARM), T, TEND, TOL, Y(N)
!                                 SPECIFICATIONS FOR SUBROUTINES
!                                 SPECIFICATIONS FOR FUNCTIONS
      EXTERNAL   FCN
!
      CALL UMACH (2, NOUT)
!                                 Set initial conditions
      T = 0.0
      Y(1) = 1.0
      Y(2) = 0.0
!                                 Set error tolerance
      TOL = 0.001
!                                 Set PARAM to default
      PARAM = 0.0E0
!                                 Select absolute error control
      PARAM(10) = 1.0
!                                 Print header
      WRITE (NOUT,99998)
      IDO = 1
      ISTEP = 0
   10 CONTINUE
      ISTEP = ISTEP + 24
      TEND = ISTEP
      CALL IVPRK (IDO, FCN, T, TEND, Y, TOL=TOL, PARAM=PARAM)
      IF (ISTEP .LE. 240) THEN
         WRITE (NOUT,'(I6,3F12.3)') ISTEP/24, T, Y
!                                 Final call to release workspace
         IF (ISTEP .EQ. 240) IDO = 3
         GO TO 10
      END IF
!                                 Show number of function calls.
      WRITE (NOUT,99999) PARAM(35)
99998 FORMAT (4X, 'ISTEP', 5X, 'Time', 9X, 'Y1', 11X, 'Y2')
99999 FORMAT (4X, 'Number of fcn calls with IVPRK =', F6.0)
      END
      SUBROUTINE FCN (N, T, Y, YPRIME)
!                                 SPECIFICATIONS FOR ARGUMENTS
```

```
      INTEGER    N
      REAL       T, Y(N), YPRIME(N)
!                                  SPECIFICATIONS FOR DATA VARIABLES
      REAL       AK1, AK2, AK3
!
      DATA AK1, AK2, AK3/294.0E0, 3.0E0, 0.01020408E0/
!
      YPRIME(1) = -Y(1) - Y(1)*Y(2) + AK1*Y(2)
      YPRIME(2) = -AK2*Y(2) + AK3*(1.0E0-Y(2))*Y(1)
      RETURN
      END
```

Output

```
ISTEP     Time          Y1          Y2
 1      24.000       0.688       0.002
 2      48.000       0.634       0.002
 3      72.000       0.589       0.002
 4      96.000       0.549       0.002
 5     120.000       0.514       0.002
 6     144.000       0.484       0.002
 7     168.000       0.457       0.002
 8     192.000       0.433       0.001
 9     216.000       0.411       0.001
10     240.000       0.391       0.001
Number of fcn calls with IVPRK = 2153.
```

IVMRK

Solves an initial-value problem $y' = f(t, y)$ for ordinary differential equations using Runge-Kutta pairs of various orders.

Required Arguments

IDO — Flag indicating the state of the computation. (Input/Output)

IDO	**State**
1	Initial entry
2	Normal re-entry
3	Final call to release workspace
4	Return after a step
5	Return for function evaluation (reverse communication)

Normally, the initial call is made with `IDO` = 1. The routine then sets `IDO` = 2, and this value is used for all but the last call that is made with `IDO` = 3. This final call is used to release workspace, which was automatically allocated by the initial call with `IDO` = 1.

FCN — User-supplied `SUBROUTINE` to evaluate functions. The usage is `CALL FCN (N, T, Y, YPRIME)`, where

N — Number of equations. (Input)
T — Independent variable. (Input)
Y — Array of size N containing the dependent variable values, *y*. (Input)
YPRIME — Array of size N containing the values of the vector *y*′ evaluated at (*t*, *y*). (Output)
FCN must be declared EXTERNAL in the calling program.

T — Independent variable. (Input/Output)
On input, T contains the initial value. On output, T is replaced by TEND unless error conditions have occurred.

TEND — Value of *t* where the solution is required. (Input)
The value of TEND may be less than the initial value of *t*.

Y — Array of size N of dependent variables. (Input/Output)
On input, Y contains the initial values. On output, Y contains the approximate solution.

YPRIME — Array of size N containing the values of the vector *y*′ evaluated at (*t*, *y*). (Output)

Optional Arguments

N — Number of differential equations. (Input)
Default: N= size (Y,1).

FORTRAN 90 Interface

Generic: `CALL IVMRK (IDO, FCN, T, TEND, Y, YPRIME [,…])`

Specific: The specific interface names are S_IVMRK and D_IVMRK.

FORTRAN 77 Interface

Single: `CALL IVMRK (IDO, N, FCN, T, TEND, Y, YPRIME)`

Double: The double precision name is DIVMRK.

Example 1

This example integrates the small system (A.2.B2) from the test set of Enright and Pryce (1987):

$$
\begin{aligned}
y_1' &= -y_1 + y_2 \\
y_2' &= y_1 - 2y_2 + y_3 \\
y_3' &= y_2 - y_3
\end{aligned}
$$

$$
\begin{aligned}
y_1(0) &= 2 \\
y_2(0) &= 0 \\
y_3(0) &= 1
\end{aligned}
$$

```
      USE IVMRK_INT
      USE WRRRN_INT
      INTEGER    N

      PARAMETER  (N=3)
!                                  Specifications for local variables
      INTEGER    IDO
      REAL       T, TEND, Y(N), YPRIME(N)
      EXTERNAL FCN
!                                  Set initial conditions
      T = 0.0
      TEND = 20.0
      Y(1) = 2.0
      Y(2) = 0.0
      Y(3) = 1.0
      IDO = 1
      CALL IVMRK (IDO, FCN, T, TEND, Y, YPRIME)
!
!                                  Final call to release workspace
      IDO = 3
      CALL IVMRK (IDO, FCN, T, TEND, Y, YPRIME)
!
      CALL WRRRN ('Y', Y)
      END
!
      SUBROUTINE FCN (N, T, Y, YPRIME)
!                                  Specifications for arguments
      INTEGER    N
      REAL       T, Y(*), YPRIME(*)
!
      YPRIME(1) = -Y(1) + Y(2)
      YPRIME(2) = Y(1) - 2.0*Y(2) + Y(3)
      YPRIME(3) = Y(2) - Y(3)
      RETURN
      END
```

Output

```
        Y
1   1.000
2   1.000
3   1.000
```

Comments

1. Workspace may be explicitly provided, if desired, by use of I2MRK/DI2MRK. The reference is:

```
CALL I2MRK (IDO, N, FCN, T, TEND, Y, YPRIME, TOL, THRES, PARAM,
YMAX, RMSERR, WORK, IWORK)
```

The additional arguments are as follows:

TOL — Tolerance for error control. (Input)

THRES — Array of size N. (Input)
THRES(I) is a threshold for solution component Y(I). It is chosen so that the value of Y(L) is not important when Y(L) is smaller in magnitude than THRES(L). THRES(L) must be greater than or equal to sqrt(amach(4)).

PARAM — A floating-point array of size 50 containing optional parameters. (Input/Output)
If a parameter is zero, then a default value is used. These default values are given below. The following parameters must be set by the user:

PARAM	**Meaning**
1 HINIT	Initial value of the step size. Must be chosen such that 0.01 ≥ HINIT ≥ 10.0 amach(4). Default: automatic selection of stepsize.
2 METHOD	Specify which Runge-Kutta pair is to be used. 1 - use the (2, 3) pair 2 - use the (4, 5) pair 3 - use the (7, 8) pair. Default: METHOD = 1 if 10-2 ≥ tol > 10-4 METHOD = 2 if 10-4 ≥ tol > 10-6 METHOD = 3 if 10-6 ≥ tol
3 ERREST	ERREST = 1 attempts to assess the true error, the difference between the numerical solution and the true solution. The cost of this is roughly twice the cost of the integration itself with METHOD = 2 or METHOD = 3, and three times with METHOD = 1. Default: ERREST = 0.
4 INTRP	If nonzero, then return the IDO = 4 before each step. See Comment 3. Default: 0
5 RCSTAT	If nonzero, then reverse communication is used to get derivative information. See Comment 4. Default: 0.
6 - 30	Not used

The following entries are set by the program:
31 HTRIAL Current trial step size.

32 NSTEP Number of steps taken.
33 NFCN Number of function evaluations.
34 ERRMAX The maximum approximate weighted true error taken over all solution components and all steps from T through the current integration point.
35 TERRMX First value of the independent variable where an approximate true error attains the maximum value ERRMAX.

YMAX Array of size N, where YMAX(L) is the largest value of ABS(Y(L)) computed at any step in the integration so far.

RMSERR — Array of size N where RMSERR(L) approximates the RMS average of the true error of the numerical solution for the L-th solution component, L = 1,..., *N*. The average is taken over all steps from T through the current integration point. RMSERR is accessed and set only if PARAM(3) = 1.

WORK — Floating point work array of size 39N using the working precision. The contents of WORK must not be changed from the first call with IDO = 1 until after the final call with IDO = 3.

IWORK — Length of array work. (Input)

2. Informational errors

Type	Code	
4	1	It does not appear possible to achieve the accuracy specified by TOL and THRES(*) using the current precision and METHOD. A larger value for METHOD, if possible, will permit greater accuracy with this precision. The integration must be restarted.
4	2	The global error assessment may not be reliable beyond the current integration point T. This may occur because either too little or too much accuracy has been requested or because *f*(*t*, *y*) is not smooth enough for values of *t* just past TEND and current values of the solution *y*. This return does not mean that you cannot integrate past TEND, rather that you cannot do it with PARAM(3) = 1.

3 If PARAM(4) is nonzero, the subroutine returns with IDO = 4 and will resume calculation at the point of interruption if re-entered with IDO = 4. Some parameters the user might want to examine are IDO, HTRIAL, NSTEP, NFCN, T, and Y. The array Y contains the newly computed trial value for *y*(*t*), accepted or not.

4 If PARAM(5) is nonzero, the subroutine will return with IDO = 5. At this time, evaluate the derivatives at T, place the result in YPRIME, and call IVMRK again. The dummy function I40RK/DI40RK may be used in place of FCN.

Description

Routine IVMRK finds an approximation to the solution of a system of first-order differential equations of the form $y' = f(t, y)$ with given initial data. Relative local error is controlled according to a user-supplied tolerance. For added efficiency, three Runge-Kutta formula pairs, of orders 3, 5, and 8, are available.

Optionally, the values of the vector y' can be passed to IVMRK by reverse communication, avoiding the user-supplied subroutine FCN. Reverse communication is especially useful in applications that have complicated algorithmic requirement for the evaluations of $f(t, y)$. Another option allows assessment of the global error in the integration.

The routine IVMRK is based on the codes contained in RKSUITE, developed by R. W. Brankin, I. Gladwell, and L. F. Shampine (1991).

Additional Examples

Example 2

This problem is the same mildly stiff problem (A.1.F2) from the test set of Enright and Pryce as Example 2 for IVPRK, page 837.

$$\begin{aligned} y_1' &= -y_1 - y_1 y_2 + k_1 y_2 \\ y_2' &= -k_2 y_2 + k_3 \left(1 - y_2\right) y_1 \\ y_1\left(0\right) &= 1 \\ y_2\left(0\right) &= 0 \\ k_1 &= 294 \\ k_2 &= 3 \\ k_3 &= 0.01020408 \\ tend &= 240 \end{aligned}$$

Although not a stiff solver, one notes the greater efficiency of IVMRK over IVPRK, in terms of derivative evaluations. Reverse communication is also used in this example. Users will find this feature particularly helpful if their derivative evaluation scheme is difficult to isolate in a separate subroutine.

```
      USE I2MRK_INT
      USE UMACH_INT
      USE AMACH_INT
      INTEGER    N

      PARAMETER  (N=2)
!                                  Specifications for local variables
      INTEGER    IDO, ISTEP, LWORK, NOUT
      REAL       PARAM(50), PREC, RMSERR(N), T, TEND, THRES(N), TOL, &
                  WORK(1000), Y(N), YMAX(N), YPRIME(N)
      REAL       AK1, AK2, AK3
      SAVE       AK1, AK2, AK3
!                                  Specifications for intrinsics
      INTRINSIC  SQRT
      REAL       SQRT
!                                  Specifications for subroutines
      EXTERNAL   I40RK
!                                  Specifications for functions
!
      DATA AK1, AK2, AK3/294.0, 3.0, 0.01020408/
!
      CALL UMACH (2, NOUT)
!                                  Set initial conditions
      T    = 0.0
      Y(1) = 1.0
      Y(2) = 0.0
!                                  Set tolerance for error control,
!                                  threshold vector and parameter
!                                  vector
      TOL = .001
      PREC = AMACH(4)
      THRES = SQRT (PREC)
      PARAM = 0.0E0
      LWORK = 1000
!                                  Turn on derivative evaluation by
!                                  reverse communication
      PARAM(5) = 1
      IDO      = 1
      ISTEP    = 24
!                                  Print header
      WRITE (NOUT,99998)
   10 CONTINUE
      TEND = ISTEP
      CALL I2MRK (IDO, N, I40RK, T, TEND, Y, YPRIME, TOL, THRES, PARAM,&
                 YMAX, RMSERR, WORK, LWORK)
      IF (IDO .EQ. 5) THEN
!                                  Evaluate derivatives
!
         YPRIME(1) = -Y(1) - Y(1)*Y(2) + AK1*Y(2)
         YPRIME(2) = -AK2*Y(2) + AK3*(1.0-Y(2))*Y(1)
         GO TO 10
      ELSE IF (ISTEP .LE. 240) THEN
!
!                                  Integrate to 10 equally spaced points
!
         WRITE (NOUT,'(I6,3F12.3)') ISTEP/24, T, Y
         IF (ISTEP .EQ. 240) IDO = 3
         ISTEP = ISTEP + 24
         GO TO 10
      END IF
!                                  Show number of derivative evaluations
!
      WRITE (NOUT,99999) PARAM(33)
99998 FORMAT (3X, 'ISTEP', 5X, 'TIME', 9X, 'Y1', 10X, 'Y2')
99999 FORMAT (/, 4X, 'NUMBER OF DERIVATIVE EVALUATIONS WITH IVMRK =', &
            F6.0)
      END

!     DUMMY FUNCTION TO TAKE THE PLACE OF DERIVATIVE EVALUATOR
```

```
      SUBROUTINE I40RK (N, T, Y, YPRIME)
      INTEGER N
      REAL     T, y(*), YPRIME(*)
      RETURN
      END
```

Output

```
ISTEP     TIME          Y1          Y2
1       24.000       0.688       0.002
2       48.000       0.634       0.002
3       72.000       0.589       0.002
4       96.000       0.549       0.002
5      120.000       0.514       0.002
6      144.000       0.484       0.002
7      168.000       0.457       0.002
8      192.000       0.433       0.001
9      216.000       0.411       0.001
10     240.000       0.391       0.001
NUMBER OF DERIVATIVE EVALUATIONS WITH IVMRK = 1375.
```

Example 3

This example demonstrates how exceptions may be handled. The problem is from Enright and Pryce (A.2.F1), and has discontinuities. We choose this problem to force a failure in the global error estimation scheme, which requires some smoothness in *y*. We also request an initial relative error tolerance which happens to be unsuitably small in this precision.

If the integration fails because of problems in global error assessment, the assessment option is turned off, and the integration is restarted. If the integration fails because the requested accuracy is not achievable, the tolerance is increased, and global error assessment is requested. The reason error assessment is turned on is that prior assessment failures may have been due more in part to an overly stringent tolerance than lack of smoothness in the derivatives.

When the integration is successful, the example prints the final relative error tolerance, and indicates whether or not global error estimation was possible.

$$y_1' = y_2$$

$$y_2' = \begin{cases} 2ay_2 - \left(\pi^2 + a^2\right) y_1 + 1, \lfloor x \rfloor \text{even} \\ 2ay_2 - \left(\pi^2 + a^2\right) y_1 - 1, \lfloor x \rfloor \text{odd} \end{cases}$$

$$y_1\left(0\right) = 0$$

$$y_2\left(0\right) = 0$$

$$a = 0.1$$

$$\lfloor x \rfloor = \text{largest integer } \le x$$

```
      USE IMSL_LIBRARIES
      INTEGER    N
      PARAMETER  (N=2)
!                                 Specifications for local variables
      INTEGER    IDO, LWORK, NOUT
      REAL       PARAM(50), PREC, RMSERR(N), T, TEND, THRES(N), TOL,&
```

```
                WORK(100), Y(N), YMAX(N), YPRIME(N)
!
                                  Specifications for intrinsics
      INTRINSIC  SQRT
      REAL       SQRT
!                                  Specifications for subroutines
!
!
!                                  Specifications for functions
      EXTERNAL   FCN
!
!
      CALL UMACH (2, NOUT)
!                                  Turn off stopping for FATAL errors
      CALL ERSET (4, -1, 0)
!                                  Initialize input, turn on global
!                                  error assessment
      LWORK = 100
      PREC = AMACH(4)
      TOL   = SQRT(PREC)
      PARAM = 0.0E01
      THRES = TOL
      TEND     = 20.0E0
      PARAM(3) = 1
!
   10 CONTINUE
!                                  Set initial values
      T    = 0.0E0
      Y(1) = 0.0E0
      Y(2) = 0.0E0
      IDO  = 1
      CALL I2MRK (IDO, N, FCN, T, TEND, Y, YPRIME, TOL, THRES, PARAM,&
                 YMAX, RMSERR, WORK, LWORK)
      IF (IERCD() .EQ. 32) THEN
!                                  Unable to achieve requested
!                                  accuracy, so increase tolerance.
!                                  Activate global error assessment
         TOL      = 10.0*TOL
         PARAM(3) = 1
         WRITE (NOUT,99995) TOL
         GO TO 10
      ELSE IF (IERCD() .EQ. 34) THEN
!                                  Global error assessment has failed,
!                                  cannot continue from this point,
!                                  so restart integration
         WRITE (NOUT,99996)
         PARAM(3) = 0
         GO TO 10
      END IF
!
!                                  Final call to release workspace
      IDO = 3
      CALL I2MRK (IDO, N, FCN, T, TEND, Y, YPRIME, TOL, THRES, PARAM,&
                 YMAX, RMSERR, WORK, LWORK)
!
```

```
!                                  Summarize status
      WRITE (NOUT,99997) TOL
      IF (PARAM(3) .EQ. 1) THEN
         WRITE (NOUT,99998)
      ELSE
         WRITE (NOUT,99999)
      END IF
      CALL WRRRN ('Y', Y)
!
99995 FORMAT (/, 'CHANGING TOLERANCE TO ', E9.3, ' AND RESTARTING ...'&
            , /, 'ALSO (RE)ENABLING GLOBAL ERROR ASSESSMENT', /)
99996 FORMAT (/, 'DISABLING GLOBAL ERROR ASSESSMENT AND RESTARTING ...'&
            , /)
99997 FORMAT (/, 72('-'), //, 'SOLUTION OBTAINED WITH TOLERANCE = ',&
            E9.3)
99998 FORMAT ('GLOBAL ERROR ASSESSMENT IS AVAILABLE')
99999 FORMAT ('GLOBAL ERROR ASSESSMENT IS NOT AVAILABLE')
!
      END
!
      SUBROUTINE FCN (N, T, Y, YPRIME)
      USE CONST_INT
!                                  Specifications for arguments
      INTEGER    N
      REAL       T, Y(*), YPRIME(*)
!                                  Specifications for local variables
      REAL       A
      REAL       PI
      LOGICAL    FIRST
      SAVE       FIRST, PI
!                                  Specifications for intrinsics
      INTRINSIC  INT, MOD
      INTEGER    INT, MOD
!                                  Specifications for functions
!
      DATA FIRST/.TRUE./
!
      IF (FIRST) THEN
         PI    = CONST('PI')
         FIRST = .FALSE.
      END IF
!
      A         = 0.1E0
      YPRIME(1) = Y(2)
      IF (MOD(INT(T),2) .EQ. 0) THEN
         YPRIME(2) = 2.0E0*A*Y(2) - (PI*PI+A*A)*Y(1) + 1.0E0
      ELSE
         YPRIME(2) = 2.0E0*A*Y(2) - (PI*PI+A*A)*Y(1) - 1.0E0
      END IF
      RETURN
      END
```

Output

```
 *** FATAL    ERROR 34 from i2mrk.  The global error assessment may not
 ***          be reliable for T past 9.994749E-01.  The integration is
 ***          being terminated.


DISABLING GLOBAL ERROR ASSESSMENT AND RESTARTING ...


 *** FATAL    ERROR 32 from i2mrk.  In order to satisfy the error
 ***          requirement I6MRK would have to use a step size of
 ***          3.647129E- 06 at TNOW = 9.999932E-01.  This is too small
 ***          for the current precision.


CHANGING TOLERANCE TO 0.345E-02 AND RESTARTING ...
ALSO (RE)ENABLING GLOBAL ERROR ASSESSMENT


 *** FATAL    ERROR 34 from i2mrk.  The global error assessment may
 ***          not be reliable for T past 9.986024E-01.  The integration
 ***          is being terminated.

DISABLING GLOBAL ERROR ASSESSMENT AND RESTARTING ...


-----------------------------------------------------------------------

SOLUTION OBTAINED WITH TOLERANCE = 0.345E-02
GLOBAL ERROR ASSESSMENT IS NOT AVAILABLE

     Y
 1  -12.30
 2    0.95
```

IVPAG

Solves an initial-value problem for ordinary differential equations using either Adams-Moulton's or Gear's BDF method.

Required Arguments

IDO — Flag indicating the state of the computation. (Input/Output)

`IDO`	**State**
1	Initial entry
2	Normal re-entry
3	Final call to release workspace

4 Return because of interrupt 1

5 Return because of interrupt 2 with step accepted

6 Return because of interrupt 2 with step rejected

7 Return for new value of matrix A.

Normally, the initial call is made with IDO = 1. The routine then sets IDO = 2, and this value is then used for all but the last call that is made with IDO = 3. This final call is only used to release workspace, which was automatically allocated by the initial call with IDO = 1. See Comment 5 for a description of the interrupts.

When IDO = 7, the matrix *A* at *t* must be recomputed and IVPAG/DIVPAG called again. No other argument (including IDO) should be changed. This value of IDO is returned only if PARAM(19) = 2.

FCN — User-supplied SUBROUTINE to evaluate functions. The usage is
CALL FCN (N, T, Y, YPRIME), where
N – Number of equations. (Input)
T – Independent variable, *t*. (Input)
Y – Array of size N containing the dependent variable values, *y*. (Input)
YPRIME – Array of size N containing the values of the vector *y*′ evaluated at (*t*, *y*). (Output)
See Comment 3.
FCN must be declared EXTERNAL in the calling program.

FCNJ — User-supplied SUBROUTINE to compute the Jacobian. The usage is
CALL FCNJ (N, T, Y, DYPDY) where
N – Number of equations. (Input)
T – Independent variable, *t*. (Input)
Y – Array of size N containing the dependent variable values, *y*(*t*). (Input)
DYPDY – An array, with data structure and type determined by PARAM(14) = MTYPE, containing the required partial derivatives $\partial f_i/\partial y_j$. (Output) These derivatives are to be evaluated at the current values of (*t*, *y*). When the Jacobian is dense, MTYPE = 0 or = 2, the leading dimension of DYPDY has the value N. When the Jacobian matrix is banded, MTYPE = 1, and the leading dimension of DYPDY has the value 2 * NLC + NUC + 1. If the matrix is banded positive definite symmetric, MTYPE = 3, and the leading dimension of DYPDY has the value NUC + 1.
FCNJ must be declared EXTERNAL in the calling program. If PARAM(19) = IATYPE is nonzero, then FCNJ should compute the Jacobian of the righthand side of the equation $Ay' = f(t, y)$. The subroutine FCNJ is used only if PARAM(13) = MITER = 1.

T — Independent variable, *t*. (Input/Output)
On input, `T` contains the initial independent variable value. On output, `T` is replaced by `TEND` unless error or other normal conditions arise. See `IDO` for details.

TEND — Value of $t = tend$ where the solution is required. (Input)
The value *tend* may be less than the initial value of *t*.

Y — Array of size `NEQ` of dependent variables, $y(t)$. (Input/Output)
On input, `Y` contains the initial values, $y(t_0)$. On output, `Y` contains the approximate solution, $y(t)$.

Optional Arguments

NEQ— Number of differential equations. (Input)
Default: `NEQ` = size (`Y`,1)

A — Matrix structure used when the system is implicit. (Input)
The matrix *A* is referenced only if `PARAM`(19) = `IATYPE` is nonzero. Its data structure is determined by `PARAM`(14) = `MTYPE`. The matrix A must be nonsingular and `MITER` must be 1 or 2. See Comment 3.

TOL — Tolerance for error control. (Input)
An attempt is made to control the norm of the local error such that the global error is proportional to `TOL`.
Default: `TOL` = .001

PARAM — A *floating-point* array of size 50 containing optional parameters. (Input/Output)
If a parameter is zero, then the default value is used. These default values are given below. Parameters that concern values of the step size are applied in the direction of integration. The following parameters may be set by the user:

	PARAM	Meaning
1	`HINIT`	Initial value of the step size `H`. Always nonnegative. Default: $0.001\|tend - t_0\|$.
2	`HMIN`	Minimum value of the step size `H`. Default: 0.0.
3	`HMAX`	Maximum value of the step size `H`. Default: No limit, beyond the machine scale, is imposed on the step size.
4	`MXSTEP`	Maximum number of steps allowed. Default: 500.
5	`MXFCN`	Maximum number of function evaluations allowed. Default: No enforced limit.
6	`MAXORD`	Maximum order of the method. Default: If Adams-Moulton method is used, then 12. If Gear’s or BDF method is used, then 5. The defaults are the maximum values allowed.
7	`INTRP1`	If this value is set nonzero, the subroutine will return before every step with `IDO` = 4. See Comment 5. Default: 0.

8	`INTRP2`	If this value is nonzero, the subroutine will return after every successful step with `IDO` = 5 and return with `IDO` = 6 after every unsuccessful step. See Comment 5. Default: 0
9	`SCALE`	A measure of the scale of the problem, such as an approximation to the average value of a norm of the Jacobian along the solution. Default: 1.0
10	`INORM`	Switch determining error norm. In the following, e_i is the absolute value of an estimate of the error in $y_i(t)$. Default: 0. 0 — min(absolute error, relative error) = max(e_i/w_i); $i = 1, \ldots, N$, where $w_i = \max(\lvert y_i(t)\rvert, 1.0)$. 1 — absolute error = max(e_i), $i = 1 \ldots, NEQ$. 2 — max(e_i / w_i), $i = 1 \ldots, N$ where $w_i = \max(\lvert y_i(t)\rvert,$ `FLOOR`), and `FLOOR` is the value `PARAM`(11). 3 — Scaled Euclidean norm defined as $$\mathrm{YMAX} = \sqrt{\sum_{i=1}^{NEQ} e_i^2 / w_i^2}$$ where $w_i = \max(\lvert y_i(t)\rvert, 1.0)$. Other definitions of `YMAX` can be specified by the user, as explained in Comment 1.
11	`FLOOR`	Used in the norm computation associated the parameter `INORM`. Default: 1.0.
12	`METH`	Integration method indicator. 1 = `METH` selects the Adams-Moulton method. 2 = `METH` selects Gear's BDF method. Default: 1.
13	`MITER`	Nonlinear solver method indicator. *Note:* If the problem is stiff and a chord or modified Newton method is most efficient, use `MITER` = 1 or = 2. 0 = `MITER` selects functional iteration. The value `IATYPE` must be set to zero with this option. 1 = `MITER` selects a chord method with a user-provided Jacobian. 2 = `MITER` selects a chord method with a divided-difference Jacobian. 3 = `MITER` selects a chord method with the Jacobian replaced by a diagonal matrix based on a directional derivative. The value `IATYPE` must be set to zero with this option. Default: 0.

14 `MTYPE` Matrix type for *A* (if used) and the Jacobian (if `MITER` = 1 or = 2). When both are used, *A* and the Jacobian must be of the same type.

0 = `MTYPE` selects full matrices.

1 = `MTYPE` selects banded matrices.

2 = `MTYPE` selects symmetric positive definite matrices.

3 = `MTYPE` selects banded symmetric positive definite matrices.

Default: 0.

15 `NLC` Number of lower codiagonals, used if `MTYPE` = 1.

Default: 0.

16 `NUC` Number of upper codiagonals, used if `MTYPE` = 1 or `MTYPE` = 3.

Default: 0.

17 Not used.

18 `EPSJ` Relative tolerance used in computing divided difference Jacobians.

Default: `SQRT`(`AMACH`(4)) .

19 `IATYPE` Type of the matrix *A*.

0 = `IATYPE` implies *A* is not used (the system is explicit).

1 = `IATYPE` if *A* is a constant matrix.

2 = `IATYPE` if *A* depends on *t*.

Default: 0.

20 `LDA` Leading dimension of array *A* exactly as specified in the dimension statement in the calling program. Used if `IATYPE` is not zero.

Default:

`N`	if `MTYPE` = 0 or = 2
`NUC` + `NLC` + 1	if `MTYPE` = 1
`NUC` + 1	if `MTYPE` = 3

21–30 Not used.

The following entries in the array `PARAM` are set by the program:

	PARAM	**Meaning**
31	`HTRIAL`	Current trial step size.
32	`HMINC`	Computed minimum step size.
33	`HMAXC`	Computed maximum step size.
34	`NSTEP`	Number of steps taken.

35	NFCN	Number of function evaluations used.
36	NJE	Number of Jacobian evaluations.
37–50		Not used.

FORTRAN 90 Interface

Generic: `CALL IVPAG (IDO, FCN, FCNJ, T, TEND, Y [,…])`

Specific: The specific interface names are `S_IVPAG` and `D_IVPAG`.

FORTRAN 77 Interface

Single: `CALL IVPAG (IDO, NEQ, FCN, FCNJ, A, T, TEND, TOL, PARAM, Y)`

Double: The double precision name is `DIVPAG`.

Example 1

Euler's equation for the motion of a rigid body not subject to external forces is

$$\begin{aligned} y_1' &= y_2 y_3 & y_1(0) &= 0 \\ y_2' &= -y_1 y_3 & y_2(0) &= 1 \\ y_3' &= -0.51 y_1 y_2 & y_3(0) &= 1 \end{aligned}$$

Its solution is, in terms of Jacobi elliptic functions, $y_1(t) = \mathrm{sn}(t; k)$, $y_2(t) = \mathrm{cn}(t; k)$, $y_3(t) = \mathrm{dn}(t; k)$ where $k^2 = 0.51$. The Adams-Moulton method of IVPAG is used to solve this system, since this is the default. All parameters are set to defaults.

The last call to IVPAG with IDO = 3 releases IMSL workspace that was reserved on the first call to IVPAG. It is not necessary to release the workspace in this example because the program ends after solving a single problem. The call to release workspace is made as a model of what would be needed if the program included further calls to IMSL routines.

Because PARAM(13) = MITER = 0, functional iteration is used and so subroutine FCNJ is never called. It is included only because the calling sequence for IVPAG requires it.

```
      USE IVPAG_INT
      USE UMACH_INT
      INTEGER    N, NPARAM
      PARAMETER  (N=3, NPARAM=50)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    IDO, IEND, NOUT
      REAL       A(1,1), T, TEND, TOL, Y(N)
!                                 SPECIFICATIONS FOR SUBROUTINES
!                                 SPECIFICATIONS FOR FUNCTIONS
      EXTERNAL   FCN, FCNJ
!                                 Initialize
!
      IDO  = 1
      T    = 0.0
      Y(1) = 0.0
      Y(2) = 1.0
      Y(3) = 1.0
      TOL  = 1.0E-6
!                                 Write title
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99998)
!                                 Integrate ODE
      IEND = 0
   10 CONTINUE
      IEND = IEND + 1
      TEND = IEND
!                                 The array a(*,*) is not used.
      CALL IVPAG (IDO, FCN, FCNJ, T, TEND, Y, TOL=TOL)
      IF (IEND .LE. 10) THEN
         WRITE (NOUT,99999) T, Y
!                                 Finish up
         IF (IEND .EQ. 10) IDO = 3
         GO TO 10
      END IF
99998 FORMAT (11X, ’T’, 14X, ’Y(1)’, 11X, ’Y(2)’, 11X, ’Y(3)’)
99999 FORMAT (4F15.5)
      END
!
      SUBROUTINE FCN (N, X, Y, YPRIME)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    N
      REAL       X, Y(N), YPRIME(N)
!
      YPRIME(1) = Y(2)*Y(3)
      YPRIME(2) = -Y(1)*Y(3)
      YPRIME(3) = -0.51*Y(1)*Y(2)
      RETURN
      END
!
      SUBROUTINE FCNJ (N, X, Y, DYPDY)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    N
      REAL       X, Y(N), DYPDY(N,*)
!                                 This subroutine is never called
      RETURN
      END
```

Output

```
     T              Y(1)           Y(2)           Y(3)
 1.00000        0.80220        0.59705        0.81963
 2.00000        0.99537       -0.09615        0.70336
 3.00000        0.64141       -0.76720        0.88892
 4.00000       -0.26961       -0.96296        0.98129
 5.00000       -0.91173       -0.41079        0.75899
 6.00000       -0.95751        0.28841        0.72967
 7.00000       -0.42877        0.90342        0.95197
 8.00000        0.51092        0.85963        0.93106
 9.00000        0.97567        0.21926        0.71730
10.00000        0.87790       -0.47884        0.77906
```

Comments

1. Workspace and a user-supplied error norm subroutine may be explicitly provided, if desired, by use of I2PAG/DI2PAG. The reference is:

```
CALL I2PAG (IDO, NEQ, FCN, FCNJ, A, T, TEND, TOL, PARAM, Y,
YTEMP, YMAX, ERROR, SAVE1, SAVE2, PW, IPVT, VNORM)
```

None of the additional array arguments should be changed from the first call with IDO = 1 until after the final call with IDO = 3. The additional arguments are as follows:

YTEMP — Array of size NMETH. (Workspace)

YMAX — Array of size NEQ containing the maximum Y-values computed so far. (Output)

ERROR — Array of size NEQ containing error estimates for each component of Y. (Output)

SAVE1 — Array of size NEQ. (Workspace)

SAVE2 — Array of size NEQ. (Workspace)

PW — Array of size NPW. (Workspace)

IPVT — Array of size NEQ. (Workspace)

VNORM — A Fortran SUBROUTINE to compute the norm of the error. (Input)
The routine may be provided by the user, or the IMSL routine I3PRK/DI3PRK may be used. In either case, the name must be declared in a Fortran ENTERNAL statement. If usage of the IMSL routine is intended, then the name I3PRK/DI3PRK should be specified. The usage of the error norm routine is CALL VNORM (NEQ, V, Y, YMAX, ENORM) where

Arg.	**Definition**
NEQ	Number of equations. (Input)
V	Array of size N containing the vector whose norm is to be computed. (Input)
Y	Array of size N containing the values of the dependent variable. (Input)
YMAX	Array of size N containing the maximum values of $\|y\ (t)\|$. (Input)
ENORM	Norm of the vector V. (Output)

VNORM must be declared EXTERNAL in the calling program.

2. Informational errors

Type	Code	
4	1	After some initial success, the integration was halted by repeated error-test failures.
4	2	The maximum number of function evaluations have been used.
4	3	The maximum number of steps allowed have been used. The problem may be stiff.
4	4	On the next step `T` + `H` will equal `T`. Either `TOL` is too small, or the problem is stiff. Note: If the Adams-Moulton method is the one used in the integration, then users can switch to the BDF methods. If the BDF methods are being used, then these comments are gratuitous and indicate that the problem is too stiff for this combination of method and value of `TOL`.
4	5	After some initial success, the integration was halted by a test on `TOL`.
4	6	Integration was halted after failing to pass the error test even after dividing the initial step size by a factor of 1.0E + 10. The value `TOL` may be too small.
4	7	Integration was halted after failing to achieve corrector convergence even after dividing the initial step size by a factor of `1.0E` + 10. The value `TOL` may be too small.
4	8	`IATYPE` is nonzero and the input matrix A multiplying y' is singular.

3. Both explicit systems, of the form $y' = f(t, y)$, and implicit systems, $Ay' = f(t, y)$, can be solved. If the system is explicit, then `PARAM`(19) = 0; and the matrix A is not referenced. If the system is implicit, then `PARAM`(14) determines the data structure of the array `A`. If `PARAM`(19) = 1, then `A` is assumed to be a constant matrix. The value of `A` used on the first call (with `IDO` = 1) is saved until after a call with `IDO` = 3. The value of `A` must not be changed between these calls.
If `PARAM`(19) = 2, then the matrix is assumed to be a function of t.

4. If `MTYPE` is greater than zero, then `MITER` must equal 1 or 2.

5. If `PARAM`(7) is nonzero, the subroutine returns with `IDO`= 4 and will resume calculation at the point of interruption if re-entered with `IDO` = 4. If `PARAM`(8) is nonzero, the subroutine will interrupt immediately after decides to accept the result of the most recent trial step. The value `IDO` = 5 is returned if the routine plans to accept, or `IDO` = 6 if it plans to reject. The value `IDO` may be changed by the user (by changing `IDO` from 6 to 5) to force acceptance of a step that would otherwise be rejected. Relevant parameters to observe after return from an interrupt are `IDO`, `HTRIAL`, `NSTEP`, `NFCN`, `NJE`, `T` and `Y`. The array `Y` contains the newly computed trial value $y(t)$.

Description

The routine `IVPAG` solves a system of first-order ordinary differential equations of the form $y' = f(t, y)$ or $Ay' = f(t, y)$ with initial conditions where A is a square nonsingular matrix of order

N. Two classes of implicit linear multistep methods are available. The first is the implicit Adams-Moulton method (up to order twelve); the second uses the backward differentiation formulas BDF (up to order five). The BDF method is often called Gear's stiff method. In both cases, because basic formulas are implicit, a system of nonlinear equations must be solved at each step. The deriviative matrix in this system has the form $L = A + \eta J$ where η is a small number computed by IVPAG and *J* is the Jacobian. When it is used, this matrix is computed in the user-supplied routine FCNJ or else it is approximated by divided differences as a default. Using defaults, *A* is the identity matrix. The data structure for the matrix *L* may be identified to be real general, real banded, symmetric positive definite, or banded symmetric positive definite. The default structure for *L* is real general.

Example 2

The BDF method of IVPAG is used to solve Example 2 of IVPRK, page 837. We set PARAM(12) = 2 to designate the BDF method. A chord or modified Newton method, with the Jacobian computed by divided differences, is used to solve the nonlinear equations. Thus, we set PARAM(13) = 2. The number of evaluations of y' is printed after the last output point, showing the efficiency gained when using a stiff solver compared to using IVPRK on this problem. The number of evaluations may vary, depending on the accuracy and other arithmetic characteristics of the computer.

```
      USE IVPAG_INT
      USE UMACH_INT
      INTEGER    MXPARM, N
      PARAMETER  (MXPARM=50, N=2)
!                                 SPECIFICATIONS FOR PARAMETERS
      INTEGER    MABSE, MBDF, MSOLVE
      PARAMETER  (MABSE=1, MBDF=2, MSOLVE=2)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    IDO, ISTEP, NOUT
      REAL       A(1,1), PARAM(MXPARM), T, TEND, TOL, Y(N)
!                                 SPECIFICATIONS FOR SUBROUTINES
!                                 SPECIFICATIONS FOR FUNCTIONS
      EXTERNAL   FCN, FCNJ
!
      CALL UMACH (2, NOUT)
!                                 Set initial conditions
      T = 0.0
      Y(1) = 1.0
      Y(2) = 0.0
!                                 Set error tolerance
      TOL = 0.001
!                                 Set PARAM to defaults
      PARAM = 0.0E0
!
      PARAM(10) = MABSE
!                                 Select BDF method
      PARAM(12) = MBDF
!                                 Select chord method and
!                                 a divided difference Jacobian.
      PARAM(13) = MSOLVE
!                                 Print header
      WRITE (NOUT,99998)
      IDO = 1
      ISTEP = 0
   10 CONTINUE
      ISTEP = ISTEP + 24
      TEND = ISTEP
!                               The array a(*,*) is not used.
      CALL IVPAG (IDO, FCN, FCNJ, T, TEND, Y, TOL=TOL, &
                 PARAM=PARAM)
      IF (ISTEP .LE. 240) THEN
         WRITE (NOUT,'(I6,3F12.3)') ISTEP/24, T, Y
!                               Final call to release workspace
         IF (ISTEP .EQ. 240) IDO = 3
         GO TO 10
      END IF
!                               Show number of function calls.
      WRITE (NOUT,99999) PARAM(35)
99998 FORMAT (4X, 'ISTEP', 5X, 'Time', 9X, 'Y1', 11X, 'Y2')
99999 FORMAT (4X, 'Number of fcn calls with IVPAG =', F6.0)
      END
      SUBROUTINE FCN (N, T, Y, YPRIME)
!                               SPECIFICATIONS FOR ARGUMENTS
      INTEGER    N
      REAL       T, Y(N), YPRIME(N)
!                               SPECIFICATIONS FOR SAVE VARIABLES
      REAL       AK1, AK2, AK3
      SAVE       AK1, AK2, AK3
!
      DATA AK1, AK2, AK3/294.0E0, 3.0E0, 0.01020408E0/
!
      YPRIME(1) = -Y(1) - Y(1)*Y(2) + AK1*Y(2)
      YPRIME(2) = -AK2*Y(2) + AK3*(1.0E0-Y(2))*Y(1)
      RETURN
      END
      SUBROUTINE FCNJ (N, T, Y, DYPDY)
!                               SPECIFICATIONS FOR ARGUMENTS
      INTEGER    N
      REAL       T, Y(N), DYPDY(N,*)
!
      RETURN
      END
```

Output

```
ISTEP     Time          Y1          Y2
 1      24.000       0.689       0.002
 2      48.000       0.636       0.002
 3      72.000       0.590       0.002
 4      96.000       0.550       0.002
 5     120.000       0.515       0.002
 6     144.000       0.485       0.002
 7     168.000       0.458       0.002
 8     192.000       0.434       0.001
 9     216.000       0.412       0.001
10     240.000       0.392       0.001
Number of fcn calls with IVPAG =   73.
```

Example 3

The BDF method of IVPAG is used to solve the so-called Robertson problem:

$$\begin{aligned} y_1' &= -c_1 y_1 + c_2 y_2 y_3 & y_1(0) &= 1 \\ y_2' &= -y_1' - y_3' & y_2(0) &= 0 \\ y_3' &= c_3 y_2^2 & y_3(0) &= 0 \\ c_1 &= 0.04, c_2 = 10^4, c_3 = 3 \times 10^7 & 0 &\le t \le 10 \end{aligned}$$

Output is obtained after each unit of the independent variable. A user-provided subroutine for the Jacobian matrix is used. An absolute error tolerance of 10^{-5} is required.

```
      USE IVPAG_INT
      USE UMACH_INT
      INTEGER    MXPARM, N
      PARAMETER  (MXPARM=50, N=3)
!                                 SPECIFICATIONS FOR PARAMETERS
      INTEGER    MABSE, MBDF, MSOLVE
      PARAMETER  (MABSE=1, MBDF=2, MSOLVE=1)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    IDO, ISTEP, NOUT
      REAL       A(1,1), PARAM(MXPARM), T, TEND, TOL, Y(N)
!                                 SPECIFICATIONS FOR SUBROUTINES
!                                 SPECIFICATIONS FOR FUNCTIONS
      EXTERNAL   FCN, FCNJ
!
      CALL UMACH (2, NOUT)
!                                 Set initial conditions
      T = 0.0
      Y(1) = 1.0
      Y(2) = 0.0
      Y(3) = 0.0
!                                 Set error tolerance
      TOL = 1.0E-5
!                                 Set PARAM to defaults
      PARAM = 0.0E0

!                                 Select absolute error control
      PARAM(10) = MABSE
!                                 Select BDF method
      PARAM(12) = MBDF
!                                 Select chord method and
!                                 a user-provided Jacobian.
      PARAM(13) = MSOLVE
!                                 Print header
      WRITE (NOUT,99998)
      IDO = 1
      ISTEP = 0
   10 CONTINUE
      ISTEP = ISTEP + 1
      TEND = ISTEP
!                                 The array a(*,*) is not used.
      CALL IVPAG (IDO, FCN, FCNJ, T, TEND, Y, TOL=TOL &
                 PARAM=PARAM)
      IF (ISTEP .LE. 10) THEN
         WRITE (NOUT,'(I6,F12.2,3F13.5)') ISTEP, T, Y
!                                 Final call to release workspace
         IF (ISTEP .EQ. 10) IDO = 3
         GO TO 10
      END IF
99998 FORMAT (4X, 'ISTEP', 5X, 'Time', 9X, 'Y1', 11X, 'Y2', 11X, &
            'Y3')
      END
      SUBROUTINE FCN (N, T, Y, YPRIME)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    N
      REAL       T, Y(N), YPRIME(N)
!                                 SPECIFICATIONS FOR SAVE VARIABLES
      REAL       C1, C2, C3
      SAVE       C1, C2, C3
!
      DATA C1, C2, C3/0.04E0, 1.0E4, 3.0E7/
!
      YPRIME(1) = -C1*Y(1) + C2*Y(2)*Y(3)
      YPRIME(3) = C3*Y(2)**2
      YPRIME(2) = -YPRIME(1) - YPRIME(3)
      RETURN
      END
      SUBROUTINE FCNJ (N, T, Y, DYPDY)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    N
      REAL       T, Y(N), DYPDY(N,*)
!                                 SPECIFICATIONS FOR SAVE VARIABLES
      REAL       C1, C2, C3
      SAVE       C1, C2, C3
!                                 SPECIFICATIONS FOR SUBROUTINES
      EXTERNAL   SSET
!
      DATA C1, C2, C3/0.04E0, 1.0E4, 3.0E7/
!                                 Clear array to zero
      CALL SSET (N**2, 0.0, DYPDY, 1)
!                                 Compute partials
      DYPDY(1,1) = -C1
      DYPDY(1,2) = C2*Y(3)
      DYPDY(1,3) = C2*Y(2)
      DYPDY(3,2) = 2.0*C3*Y(2)
      DYPDY(2,1) = -DYPDY(1,1)
      DYPDY(2,2) = -DYPDY(1,2) - DYPDY(3,2)
      DYPDY(2,3) = -DYPDY(1,3)
      RETURN
      END
```

Output

```
 ISTEP     Time         Y1           Y2           Y3
 1        1.00      0.96647      0.00003      0.03350
 2        2.00      0.94164      0.00003      0.05834
 3        3.00      0.92191      0.00002      0.07806
 4        4.00      0.90555      0.00002      0.09443
 5        5.00      0.89153      0.00002      0.10845
```

```
   6         6.00      0.87928      0.00002      0.12070
   7         7.00      0.86838      0.00002      0.13160
   8         8.00      0.85855      0.00002      0.14143
   9         9.00      0.84959      0.00002      0.15039
  10        10.00      0.84136      0.00002      0.15862
```

Example 4

Solve the partial differential equation

$$e^{-t}\frac{\partial u}{\partial t}=\frac{\partial^2 u}{\partial x^2}$$

with the initial condition

$$u(t=0,x)=\sin x$$

and the boundary conditions

$$u(t,x=0)=u(t,x=\pi)=0$$

on the square [0, 1] × [0, π], using the method of lines with a piecewise-linear Galerkin discretization. The exact solution is $u(t, x) = \exp(1 - e^t) \sin x$. The interval [0, π] is divided into equal intervals by choosing breakpoints $x_k = k\pi/(N + 1)$ for $k = 0, \ldots, N + 1$. The unknown function $u(t, x)$ is approximated by

$$\sum\nolimits_{k=1}^{N} c_k(t)\phi_k(x)$$

where $\phi_k(x)$ is the piecewiselinear function that equals 1 at x_k and is zero at all of the other breakpoints. We approximate the partial differential equation by a system of N ordinary differential equations, $A\, dc/dt = Rc$ where A and R are matrices of order N. The matrix A is given by

$$A_{ij}=e^{-t}\int_0^{\pi}\phi_i(x)\,\phi_j(x)\,dx=\begin{array}{ll} e^{-t}2h/3 & \text{if } i=j \\ e^{-t}h/6 & \text{if } i=j\pm 1 \\ 0 & \text{otherwise} \end{array}$$

where $h = 1/(N + 1)$ is the mesh spacing. The matrix R is given by

$$R_{ij}=\int_0^{\pi}\phi_i''(x)\phi_j(x)\,dx=-\int_0^{\pi}\phi_i'(x)\phi_j'(x)dx=\begin{array}{ll} -2/h & \text{if } i=j \\ 1/h & \text{if } i=j\pm 1 \\ 0 & \text{otherwise} \end{array}$$

The integrals involving

$$\phi_i''$$

are assigned the values of the integrals on the right-hand side, by using the boundary values and integration by parts. Because this system may be stiff, Gear's BDF method is used.

In the following program, the array Y(1:N) corresponds to the vector of coefficients, *c*. Note that Y contains N + 2 elements; Y(0) and Y(N + 1) are used to store the boundary values. The matrix *A* depends on *t* so we set PARAM(19) = 2 and evaluate *A* when IVPAG returns with IDO = 7. The subroutine FCN computes the vector *Rc*, and the subroutine FCNJ computes *R*. The matrices *A* and *R* are stored as band-symmetric positive-definite structures having one upper co-diagonal.

```
      USE IVPAG_INT
      USE CONST_INT
      USE WRRRN_INT
      USE SSET_INT
      INTEGER    LDA, N, NPARAM, NUC
      PARAMETER  (N=9, NPARAM=50, NUC=1, LDA=NUC+1)
!                                 SPECIFICATIONS FOR PARAMETERS
      INTEGER    NSTEP
      PARAMETER  (NSTEP=4)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    I, IATYPE, IDO, IMETH, INORM, ISTEP, MITER, MTYPE
      REAL       A(LDA,N), C, HINIT, PARAM(NPARAM), PI, T, TEND, TMAX, &
                TOL, XPOINT(0:N+1), Y(0:N+1)
      CHARACTER  TITLE*10
!                                 SPECIFICATIONS FOR COMMON /COMHX/
      COMMON     /COMHX/ HX
      REAL       HX
!                                 SPECIFICATIONS FOR INTRINSICS
      INTRINSIC  EXP, REAL, SIN
      REAL       EXP, REAL, SIN
!                                 SPECIFICATIONS FOR SUBROUTINES
!                                 SPECIFICATIONS FOR FUNCTIONS
      EXTERNAL   FCN, FCNJ
!                                 Initialize PARAM
      HINIT  = 1.0E-3
      INORM  = 1
      IMETH  = 2
      MITER  = 1
      MTYPE  = 3
      IATYPE = 2
      PARAM = 0.0E0
      PARAM(1)  = HINIT
      PARAM(10) = INORM
      PARAM(12) = IMETH
      PARAM(13) = MITER
      PARAM(14) = MTYPE
      PARAM(16) = NUC
      PARAM(19) = IATYPE
!                                 Initialize other arguments
      PI = CONST(’PI’)
      HX = PI/REAL(N+1)
      CALL SSET (N-1, HX/6., A(1:,2), LDA)
      CALL SSET (N, 2.*HX/3., A(2:,1), LDA)
      DO 10  I=0, N + 1
         XPOINT(I) = I*HX
         Y(I)      = SIN(XPOINT(I))
   10 CONTINUE
      TOL  = 1.0E-6
      T    = 0.0
      TMAX = 1.0
!                                 Integrate ODE
      IDO   = 1
      ISTEP = 0
   20 CONTINUE
      ISTEP = ISTEP + 1
      TEND  = TMAX*REAL(ISTEP)/REAL(NSTEP)
   30 CALL IVPAG (IDO, FCN, FCNJ, T, TEND, Y(1:), NEQ=N, A=A, &
                  TOL=TOL, PARAM=PARAM)
!                                 Set matrix A
      IF (IDO .EQ. 7) THEN
         C = EXP(-T)
         CALL SSET (N-1, C*HX/6., A(1:,2), LDA)
         CALL SSET (N, 2.*C*HX/3., A(2:,1), LDA)
         GO TO 30
      END IF
      IF (ISTEP .LE. NSTEP) THEN
!                                 Print solution
         WRITE (TITLE,’(A,F5.3,A)’) ’U(T=’, T, ’)’
         CALL WRRRN (TITLE, Y, 1, N+2, 1)
!                                 Final call to release workspace
         IF (ISTEP .EQ. NSTEP) IDO = 3
         GO TO 20
       END IF
       END
!
      SUBROUTINE FCN (N, T, Y, YPRIME)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    N
      REAL       T, Y(*), YPRIME(N)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    I
!                                 SPECIFICATIONS FOR COMMON /COMHX/
      COMMON     /COMHX/ HX
      REAL       HX
!                                 SPECIFICATIONS FOR SUBROUTINES
      EXTERNAL   SSCAL
!
      YPRIME(1) = -2.0*Y(1) + Y(2)
      DO 10  I=2, N - 1
         YPRIME(I) = -2.0*Y(I) + Y(I-1) + Y(I+1)
   10 CONTINUE
      YPRIME(N) = -2.0*Y(N) + Y(N-1)
      CALL SSCAL (N, 1.0/HX, YPRIME, 1)
      RETURN
      END
!
      SUBROUTINE FCNJ (N, T, Y, DYPDY)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    N
      REAL       T, Y(*), DYPDY(2,*)
!                                 SPECIFICATIONS FOR COMMON /COMHX/
      COMMON     /COMHX/ HX
      REAL       HX
!                                 SPECIFICATIONS FOR SUBROUTINES
```

```
      EXTERNAL   SSET
!
      CALL SSET (N-1, 1.0/HX, DYPDY(1,2), 2)
      CALL SSET (N, -2.0/HX, DYPDY(2,1), 2)
      RETURN
      END
```

Output

```
                             U(T=0.250)
     1         2         3         4         5         6         7         8
0.0000    0.2321    0.4414    0.6076    0.7142    0.7510    0.7142    0.6076


     9        10        11
0.4414    0.2321    0.0000


                             U(T=0.500)
     1         2         3         4         5         6         7         8
0.0000    0.1607    0.3056    0.4206    0.4945    0.5199    0.4945    0.4206


     9        10        11
0.3056    0.1607    0.0000


                             U(T=0.750)
     1         2         3         4         5         6         7         8
0.0000    0.1002    0.1906    0.2623    0.3084    0.3243    0.3084    0.2623


     9        10        11
0.1906    0.1002    0.0000


                             U(T=1.000)
     1         2         3         4         5         6         7         8
0.0000    0.0546    0.1039    0.1431    0.1682    0.1768    0.1682    0.1431


     9        10        11
0.1039    0.0546    0.0000
```

BVPFD

Solves a (parameterized) system of differential equations with boundary conditions at two points, using a variable order, variable step size finite difference method with deferred corrections.

Required Arguments

FCNEQN — User-supplied `SUBROUTINE` to evaluate derivatives. The usage is `CALL FCNEQN (N, T, Y, P, DYDT)`, where

`N` – Number of differential equations. (Input)

T – Independent variable, *t*. (Input)
Y – Array of size N containing the dependent variable values, *y*(t). (Input)
P – Continuation parameter, *p*. (Input)
See Comment 3.
DYDT – Array of size N containing the derivatives $y'(t)$. (Output)
The name FCNEQN must be declared EXTERNAL in the calling program.

FCNJAC — User-supplied SUBROUTINE to evaluate the Jacobian. The usage is CALL FCNJAC (N, T, Y, P, DYPDY), where
N – Number of differential equations. (Input)
T – Independent variable, *t*. (Input)
Y – Array of size N containing the dependent variable values. (Input)
P – Continuation parameter, *p*. (Input)
See Comments 3.
DYPDY – N by N array containing the partial derivatives $a_{i,j} = \partial f_i / \partial y_j$ evaluated at (t, y). The values $a_{i,j}$ are returned in DYPDY(i, j). (Output)
The name FCNJAC must be declared EXTERNAL in the calling program.

FCNBC — User-supplied SUBROUTINE to evaluate the boundary conditions. The usage is CALL FCNBC (N, YLEFT, YRIGHT, P, H), where
N – Number of differential equations. (Input)
YLEFT – Array of size N containing the values of the dependent variable at the left endpoint. (Input)
YRIGHT – Array of size N containing the values of the dependent variable at the right endpoint. (Input)
P – Continuation parameter, *p*. (Input)
See Comment 3.
H – Array of size N containing the boundary condition residuals. (Output)

The boundary conditions are defined by $h_i = 0$; for $i = 1, \ldots, N$. The left endpoint conditions must be defined first, then, the conditions involving both endpoints, and finally the right endpoint conditions.

The name FCNBC must be declared EXTERNAL in the calling program.

FCNPEQ — User-supplied SUBROUTINE to evaluate the partial derivative of y' with respect to the parameter *p*. The usage is

CALL FCNPEQ (N, T, Y, P, DYPDP), where
N – Number of differential equations. (Input)
T – Dependent variable, *t*. (Input)
Y – Array of size N containing the dependent variable values. (Input)
P – Continuation parameter, *p*. (Input)
See Comment 3.

DYPDP – Array of size N containing the partial derivatives $a_{i,j} = \partial f_i / \partial y_j$ evaluated at (t, y). The values $a_{i,j}$ are returned in DYPDY(i, j). (Output)

The name FCNPEQ must be declared EXTERNAL in the calling program.

FCNPBC — User-supplied SUBROUTINE to evaluate the derivative of the boundary conditions with respect to the parameter *p*. The usage is
CALL FCNPBC (N, YLEFT, YRIGHT, P, H), where
N – Number of differential equations. (Input)
YLEFT – Array of size N containing the values of the dependent variable at the left endpoint. (Input)
YRIGHT – Array of size N containing the values of the dependent variable at the right endpoint. (Input)
P – Continuation parameter, *p*. (Input)
See Comment 3.
H – Array of size N containing the derivative of f_i with respect to *p*. (Output)

The name FCNPBC must be declared EXTERNAL in the calling program.

NLEFT — Number of initial conditions. (Input)
The value NLEFT must be greater than or equal to zero and less than N.

NCUPBC — Number of coupled boundary conditions. (Input)
The value NLEFT + NCUPBC must be greater than zero and less than or equal to N.

TLEFT — The left endpoint. (Input)

TRIGHT — The right endpoint. (Input)

PISTEP — Initial increment size for *p*. (Input)
If this value is zero, continuation will not be used in this problem. The routines FCNPEQ and FCNPBC will not be called.

TOL — Relative error control parameter. (Input)
The computations stop when ABS(ERROR(J, I))/MAX(ABS(Y(J, I)), 1.0).LT.TOL for all J = 1, …, N and I = 1, …, NGRID. Here, ERROR(J, I) is the estimated error in Y(J, I).

TINIT — Array of size NINIT containing the initial grid points. (Input)

YINIT — Array of size N by NINIT containing an initial guess for the values of Y at the points in TINIT. (Input)

LINEAR — Logical .TRUE. if the differential equations and the boundary conditions are linear. (Input)

MXGRID — Maximum number of grid points allowed. (Input)

NFINAL — Number of final grid points, including the endpoints. (Output)

TFINAL — Array of size `MXGRID` containing the final grid points. (Output)
Only the first `NFINAL` points are significant.

YFINAL — Array of size `N` by `MXGRID` containing the values of `Y` at the points in `TFINAL`. (Output)

ERREST — Array of size `N`. (Output)
`ERREST`(`J`) is the estimated error in `Y`(`J`).

Optional Arguments

N — Number of differential equations. (Input)
Default: `N` = size (`YINIT`,1).

NINIT — Number of initial grid points, including the endpoints. (Input)
It must be at least 4.
Default: `NINIT` = size (`TINIT`,1).

LDYINI — Leading dimension of `YINIT` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDYINI` = size (`YINIT`,1).

PRINT — Logical `.TRUE.` if intermediate output is to be printed. (Input)
Default: `PRINT` = `.FALSE.`

LDYFIN — Leading dimension of `YFINAL` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDYFIN` = size (`YFINAL`,1).

FORTRAN 90 Interface

Generic:

```
CALL BVPFD (FCNEQN, FCNJAC, FCNBC, FCNPEQ, FCNPBC, NLEFT,
NCUPBC, TLEFT, TRIGHT, PISTEP, TOL, TINIT,
YINIT, LINEAR, MXGRID, NFINAL, TFINAL, YFINAL,
ERREST [,…])
```

Specific: The specific interface names are `S_BVPFD` and `D_BVPFD`.

FORTRAN 77 Interface

Single:

```
CALL BVPFD (FCNEQN, FCNJAC, FCNBC, FCNPEQ, FCNPBC, N,
NLEFT, NCUPBC, TLEFT, TRIGHT, PISTEP, TOL, NINIT, TINIT,
YINIT, LDYINI, LINEAR, PRINT, MXGRID, NFINAL, TFINAL,
YFINAL, LDYFIN, ERREST)
```

Double: The double precision name is DBVPFD.

Example 1

This example solves the third-order linear equation

$$y''' - 2y'' + y' - y = \sin t$$

subject to the boundary conditions $y(0) = y(2\pi)$ and $y'(0) = y'(2\pi) = 1$. (Its solution is $y = \sin t$.) To use BVPFD, the problem is reduced to a system of first-order equations by defining $y_1 = y, y_2 = y'$ and $y_3 = y''$. The resulting system is

$$\begin{aligned} y_1' &= y_2 & y_2(0) - 1 &= 0 \\ y_2' &= y_3 & y_1(0) - y_1(2\pi) &= 0 \\ y_3' &= 2y_3 - y_2 + y_1 + \sin t & y_2(2\pi) - 1 &= 0 \end{aligned}$$

Note that there is one boundary condition at the left endpoint $t = 0$ and one boundary condition coupling the left and right endpoints. The final boundary condition is at the right endpoint. The total number of boundary conditions must be the same as the number of equations (in this case 3).

Note that since the parameter p is not used in the call to BVPFD, the routines FCNPEQ and FCNPBC are not needed. Therefore, in the call to BVPFD, FCNEQN and FCNBC were used in place of FCNPEQ and FCNPBC.

```
      USE BVPFD_INT
      USE UMACH_INT
      USE CONST_INT
!                                  SPECIFICATIONS FOR PARAMETERS
      INTEGER    LDYFIN, LDYINI, MXGRID, NEQNS, NINIT
      PARAMETER  (MXGRID=45, NEQNS=3, NINIT=10, LDYFIN=NEQNS, &
                LDYINI=NEQNS)
!                                  SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    I, J, NCUPBC, NFINAL, NLEFT, NOUT
      REAL       ERREST(NEQNS), PISTEP, TFINAL(MXGRID), TINIT(NINIT), &
                TLEFT, TOL, TRIGHT, YFINAL(LDYFIN,MXGRID), &
                YINIT(LDYINI,NINIT)
      LOGICAL    LINEAR, PRINT
!                                  SPECIFICATIONS FOR INTRINSICS
      INTRINSIC  FLOAT
      REAL       FLOAT
!                                  SPECIFICATIONS FOR SUBROUTINES
!                                  SPECIFICATIONS FOR FUNCTIONS
      EXTERNAL   FCNBC, FCNEQN, FCNJAC
!                                  Set parameters
      NLEFT  = 1
      NCUPBC = 1
      TOL    = .001
      TLEFT  = 0.0
      TRIGHT = CONST('PI')
      TRIGHT = 2.0*TRIGHT
      PISTEP = 0.0
      PRINT  = .FALSE.
      LINEAR = .TRUE.
!                                 Define TINIT
      DO 10  I=1, NINIT
         TINIT(I) = TLEFT + (I-1)*(TRIGHT-TLEFT)/FLOAT(NINIT-1)
   10 CONTINUE
!                                 Set YINIT to zero
         YINIT = 0.0E0
!                                 Solve problem
      CALL BVPFD (FCNEQN, FCNJAC, FCNBC, FCNEQN, FCNBC, NLEFT, &
                 NCUPBC, TLEFT, TRIGHT, PISTEP, TOL, TINIT, &
                 YINIT, LINEAR, MXGRID, NFINAL, &
                 TFINAL, YFINAL, ERREST)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99997)
      WRITE (NOUT,99998) (I,TFINAL(I),(YFINAL(J,I),J=1,NEQNS),I=1, &
                       NFINAL)
      WRITE (NOUT,99999) (ERREST(J),J=1,NEQNS)
99997 FORMAT (4X, ’I’, 7X, ’T’, 14X, ’Y1’, 13X, ’Y2’, 13X, ’Y3’)
99998 FORMAT (I5, 1P4E15.6)
99999 FORMAT (’ Error estimates’, 4X, 1P3E15.6)
      END
      SUBROUTINE FCNEQN (NEQNS, T, Y, P, DYDX)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NEQNS
      REAL       T, P, Y(NEQNS), DYDX(NEQNS)
!                                 SPECIFICATIONS FOR INTRINSICS
      INTRINSIC  SIN
      REAL       SIN
!                                 Define PDE
      DYDX(1) = Y(2)
      DYDX(2) = Y(3)
      DYDX(3) = 2.0*Y(3) - Y(2) + Y(1) + SIN(T)
      RETURN
      END
      SUBROUTINE FCNJAC (NEQNS, T, Y, P, DYPDY)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NEQNS
      REAL       T, P, Y(NEQNS), DYPDY(NEQNS,NEQNS)
!                                 Define d(DYDX)/dY
      DYPDY(1,1) = 0.0
      DYPDY(1,2) = 1.0
      DYPDY(1,3) = 0.0
      DYPDY(2,1) = 0.0
      DYPDY(2,2) = 0.0
      DYPDY(2,3) = 1.0
      DYPDY(3,1) = 1.0
      DYPDY(3,2) = -1.0
      DYPDY(3,3) = 2.0
      RETURN
      END
      SUBROUTINE FCNBC (NEQNS, YLEFT, YRIGHT, P, F)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NEQNS
      REAL       P, YLEFT(NEQNS), YRIGHT(NEQNS), F(NEQNS)
!                                 Define boundary conditions
      F(1) = YLEFT(2) - 1.0
      F(2) = YLEFT(1) - YRIGHT(1)
      F(3) = YRIGHT(2) - 1.0
      RETURN
      END
```

Output

```
 I        T             Y1              Y2             Y3
 1   0.000000E+00  -1.123191E-04   1.000000E+00    6.242319E05
 2   3.490659E-01   3.419107E-01   9.397087E-01   -3.419580E01
 3   6.981317E-01   6.426908E-01   7.660918E-01   -6.427230E-01
 4   1.396263E+00   9.847531E-01   1.737333E-01   -9.847453E-01
 5   2.094395E+00   8.660529E-01  -4.998747E-01   -8.660057E-01
 6   2.792527E+00   3.421830E-01  -9.395474E-01   -3.420648E-01
 7   3.490659E+00  -3.417234E-01  -9.396111E-01    3.418948E-01
 8   4.188790E+00  -8.656880E-01  -5.000588E-01    8.658733E-01
 9   4.886922E+00  -9.845794E-01   1.734571E-01    9.847518E-01
10   5.585054E+00  -6.427721E-01   7.658258E-01    6.429526E-01
11   5.934120E+00  -3.420819E-01   9.395434E-01    3.423986E-01
12   6.283185E+00  -1.123186E-04   1.000000E+00    6.743190E-04
Error estimates     2.840430E-04   1.792939E-04    5.588399E-04
```

Comments

1. Workspace may be explicitly provided, if desired, by use of B2PFD/DB2PFD. The reference is:

   ```
   CALL B2PFD (FCNEQN, FCNJAC, FCNBC, FCNPEQ, FCNPBC, N, NLEFT,
   NCUPBC, TLEFT, TRIGHT, PISTEP, TOL, NINIT, TINIT, YINIT, LDYINI,
   LINEAR, PRINT, MXGRID, NFINAL, TFINAL, YFINAL, LDYFIN, ERREST,
   RWORK, IWORK)
   ```

   The additional arguments are as follows:

   RWORK — Floating-point work array of size `N(3N * MXGRID + 4N + 1) + MXGRID * (7N + 2)`.

   IWORK — Integer work array of size `2N * MXGRID + N + MXGRID`.

2. Informational errors

Type	Code	
4	1	More than MXGRID grid points are needed to solve the problem.
4	2	Newton's method diverged.
3	3	Newton's method reached roundoff error level.

3. If the value of PISTEP is greater than zero, then the routine BVPFD assumes that the user has embedded the problem into a one-parameter family of problems:

$$y' = y'(t, y, p)$$

$$h(y_{tleft}, y_{tright}, p) = 0$$

such that for $p = 0$ the problem is simple. For $p = 1$, the original problem is recovered. The routine BVPFD automatically attempts to increment from $p = 0$ to $p = 1$. The value PISTEP is the beginning increment used in this continuation. The increment will usually be changed by routine BVPFD, but an arbitrary minimum of 0.01 is imposed.

4. The vectors TINIT and TFINAL may be the same.

5. The arrays YINIT and YFINAL may be the same.

Description

The routine BVPFD is based on the subprogram PASVA3 by M. Lentini and V. Pereyra (see Pereyra 1978). The basic discretization is the trapezoidal rule over a nonuniform mesh. This mesh is chosen adaptively, to make the local error approximately the same size everywhere. Higher-order discretizations are obtained by deferred corrections. Global error estimates are produced to control the computation. The resulting nonlinear algebraic system is solved by Newton's method with step control. The linearized system of equations is solved by a special form of Gauss elimination that preserves the sparseness.

Example 2

In this example, the following nonlinear problem is solved:

$$y'' - y^3 + (1 + \sin^2 t) \sin t = 0$$

with $y(0) = y(\pi) = 0$. Its solution is $y = \sin t$. As in Example 1, this equation is reduced to a system of first-order differential equations by defining $y_1 = y$ and $y_2 = y'$. The resulting system is

$$\begin{aligned} y_1' &= y_2 & y_1(0) &= 0 \\ y_2' &= y_1^3 - \left(1 + \sin^2 t\right)\sin t & y_1(\pi) &= 0 \end{aligned}$$

In this problem, there is one boundary condition at the left endpoint and one at the right endpoint; there are no coupled boundary conditions.

Note that since the parameter p is not used, in the call to BVPFD the routines FCNPEQ and FCNPBC are not needed. Therefore, in the call to BVPFD, FCNEQN and FCNBC were used in place of FCNPEQ and FCNPBC.

```
      USE BVPFD_INT
      USE UMACH_INT
      USE CONST_INT

!                                  SPECIFICATIONS FOR PARAMETERS
      INTEGER    LDYFIN, LDYINI, MXGRID, NEQNS, NINIT
      PARAMETER  (MXGRID=45, NEQNS=2, NINIT=12, LDYFIN=NEQNS, &
                 LDYINI=NEQNS)
!                                  SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    I, J, NCUPBC, NFINAL, NLEFT, NOUT
      REAL       ERREST(NEQNS), PISTEP, TFINAL(MXGRID), TINIT(NINIT), &
```

```
                TLEFT, TOL, TRIGHT, YFINAL(LDYFIN,MXGRID), &
                YINIT(LDYINI,NINIT)
      LOGICAL    LINEAR, PRINT
!                                 SPECIFICATIONS FOR INTRINSICS
      INTRINSIC  FLOAT
      REAL       FLOAT
!                                 SPECIFICATIONS FOR FUNCTIONS
      EXTERNAL   FCNBC, FCNEQN, FCNJAC
!                                 Set parameters
      NLEFT  = 1
      NCUPBC = 0
      TOL    = .001
      TLEFT  = 0.0
      TRIGHT = CONST('PI')
      PISTEP = 0.0
      PRINT  = .FALSE.
      LINEAR = .FALSE.
!                                 Define TINIT and YINIT
      DO 10  I=1, NINIT
         TINIT(I)   = TLEFT + (I-1)*(TRIGHT-TLEFT)/FLOAT(NINIT-1)
         YINIT(1,I) = 0.4*(TINIT(I)-TLEFT)*(TRIGHT-TINIT(I))
         YINIT(2,I) = 0.4*(TLEFT-TINIT(I)+TRIGHT-TINIT(I))
   10 CONTINUE
!                                 Solve problem
      CALL BVPFD (FCNEQN, FCNJAC, FCNBC, FCNEQN, FCNBC, NLEFT, &
                 NCUPBC, TLEFT, TRIGHT, PISTEP, TOL, TINIT, &
                 YINIT, LINEAR, MXGRID, NFINAL, &
                 TFINAL, YFINAL, ERREST)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99997)
      WRITE (NOUT,99998) (I,TFINAL(I),(YFINAL(J,I),J=1,NEQNS),I=1, &
                   NFINAL)
      WRITE (NOUT,99999) (ERREST(J),J=1,NEQNS)
99997 FORMAT (4X, 'I', 7X, 'T', 14X, 'Y1', 13X, 'Y2')
99998 FORMAT (I5, 1P3E15.6)
99999 FORMAT (' Error estimates', 4X, 1P2E15.6)
      END
      SUBROUTINE FCNEQN (NEQNS, T, Y, P, DYDT)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NEQNS
      REAL       T, P, Y(NEQNS), DYDT(NEQNS)
!                                 SPECIFICATIONS FOR INTRINSICS
      INTRINSIC  SIN
      REAL       SIN
!                                 Define PDE
      DYDT(1) = Y(2)
      DYDT(2) = Y(1)**3 - SIN(T)*(1.0+SIN(T)**2)
      RETURN
      END
      SUBROUTINE FCNJAC (NEQNS, T, Y, P, DYPDY)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NEQNS
      REAL       T, P, Y(NEQNS), DYPDY(NEQNS,NEQNS)
!                                 Define d(DYDT)/dY
```

```
      DYPDY(1,1) = 0.0
      DYPDY(1,2) = 1.0
      DYPDY(2,1) = 3.0*Y(1)**2
      DYPDY(2,2) = 0.0
      RETURN
      END
      SUBROUTINE FCNBC (NEQNS, YLEFT, YRIGHT, P, F)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NEQNS
      REAL       P, YLEFT(NEQNS), YRIGHT(NEQNS), F(NEQNS)
!                                 Define boundary conditions
      F(1) = YLEFT(1)
      F(2) = YRIGHT(1)
      RETURN
      END
```

Output

```
 I        T               Y1               Y2
 1   0.000000E+00   0.000000E+00   9.999277E-01
 2   2.855994E-01   2.817682E-01   9.594315E-01
 3   5.711987E-01   5.406458E-01   8.412407E-01
 4   8.567980E-01   7.557380E-01   6.548904E-01
 5   1.142397E+00   9.096186E-01   4.154530E-01
 6   1.427997E+00   9.898143E-01   1.423307E-01
 7   1.713596E+00   9.898143E-01  -1.423307E-01
 8   1.999195E+00   9.096185E-01  -4.154530E-01
 9   2.284795E+00   7.557380E-01  -6.548903E-01
10   2.570394E+00   5.406460E-01  -8.412405E-01
11   2.855994E+00   2.817683E-01  -9.594313E-01
12   3.141593E+00   0.000000E+00  -9.999274E-01
Error estimates     3.906105E-05   7.124186E-05
```

Example 3

In this example, the following nonlinear problem is solved:

$$y'' - y^3 = \frac{40}{9}\left(t - \frac{1}{2}\right)^{2/3} - \left(t - \frac{1}{2}\right)^8$$

with $y(0) = y(1) = \pi/2$. As in the previous examples, this equation is reduced to a system of first-order differential equations by defining $y_1 = y$ and $y_2 = y'$. The resulting system is

$$\begin{array}{ll} y_1' = y_2 & y_1(0) = \pi/2 \\ y_2' = y_1^3 - \dfrac{40}{9}\left(t - \dfrac{1}{2}\right)^{2/3} + \left(t - \dfrac{1}{2}\right)^8 & y_1(1) = \pi/2 \end{array}$$

The problem is embedded in a family of problems by introducing the parameter p and by changing the second differential equation to

$$y_2' = py_1^3 + \frac{40}{9}\left(t-\frac{1}{2}\right)^{2/3} - \left(t-\frac{1}{2}\right)^8$$

At $p = 0$, the problem is linear; and at $p = 1$, the original problem is recovered. The derivatives $\partial y'/\partial p$ must now be specified in the subroutine FCNPEQ. The derivatives $\partial f/\partial p$ are zero in FCNPBC.

```
      USE BVPFD_INT
      USE UMACH_INT
!                                 SPECIFICATIONS FOR PARAMETERS
      INTEGER    LDYFIN, LDYINI, MXGRID, NEQNS, NINIT
      PARAMETER  (MXGRID=45, NEQNS=2, NINIT=5, LDYFIN=NEQNS, &
                LDYINI=NEQNS)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    NCUPBC, NFINAL, NLEFT, NOUT
      REAL       ERREST(NEQNS), PISTEP, TFINAL(MXGRID), TLEFT, TOL, &
                XRIGHT, YFINAL(LDYFIN,MXGRID)
      LOGICAL    LINEAR, PRINT
!                                 SPECIFICATIONS FOR SAVE VARIABLES
      INTEGER    I, J
      REAL       TINIT(NINIT), YINIT(LDYINI,NINIT)
      SAVE       I, J, TINIT, YINIT
!                                 SPECIFICATIONS FOR FUNCTIONS
      EXTERNAL   FCNBC, FCNEQN, FCNJAC, FCNPBC, FCNPEQ
!
      DATA TINIT/0.0, 0.4, 0.5, 0.6, 1.0/
      DATA ((YINIT(I,J),J=1,NINIT),I=1,NEQNS)/0.15749, 0.00215, 0.0, &
          0.00215, 0.15749, -0.83995, -0.05745, 0.0, 0.05745, 0.83995/
!                                 Set parameters
      NLEFT  = 1
      NCUPBC = 0
      TOL    = .001
      TLEFT  = 0.0
      XRIGHT = 1.0
      PISTEP = 0.1
      PRINT  = .FALSE.
      LINEAR = .FALSE.
!
      CALL BVPFD (FCNEQN, FCNJAC, FCNBC, FCNPEQ, FCNPBC, NLEFT, &
                  NCUPBC, TLEFT, XRIGHT, PISTEP, TOL, TINIT, &
                  YINIT, LINEAR, MXGRID, NFINAL,TFINAL, YFINAL, ERREST)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99997)
      WRITE (NOUT,99998) (I,TFINAL(I),(YFINAL(J,I),J=1,NEQNS),I=1, &
                       NFINAL)
      WRITE (NOUT,99999) (ERREST(J),J=1,NEQNS)
99997 FORMAT (4X, 'I', 7X, 'T', 14X, 'Y1', 13X, 'Y2')
99998 FORMAT (I5, 1P3E15.6)
99999 FORMAT (' Error estimates', 4X, 1P2E15.6)
      END
      SUBROUTINE FCNEQN (NEQNS, T, Y, P, DYDT)
!                                  SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NEQNS
      REAL       T, P, Y(NEQNS), DYDT(NEQNS)
!                                 Define PDE
      DYDT(1) = Y(2)
      DYDT(2) = P*Y(1)**3 + 40./9.*((T-0.5)**2)**(1./3.) - (T-0.5)**8
      RETURN
      END
      SUBROUTINE FCNJAC (NEQNS, T, Y, P, DYPDY)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NEQNS
      REAL       T, P, Y(NEQNS), DYPDY(NEQNS,NEQNS)
!                                 Define d(DYDT)/dY
      DYPDY(1,1) = 0.0
      DYPDY(1,2) = 1.0
      DYPDY(2,1) = P*3.*Y(1)**2
      DYPDY(2,2) = 0.0
      RETURN
      END
      SUBROUTINE FCNBC (NEQNS, YLEFT, YRIGHT, P, F)
      USE CONST_INT
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NEQNS
      REAL       P, YLEFT(NEQNS), YRIGHT(NEQNS), F(NEQNS)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      REAL       PI
!                                 Define boundary conditions
      PI   = CONST(’PI’)
      F(1) = YLEFT(1) - PI/2.0
      F(2) = YRIGHT(1) - PI/2.0
      RETURN
      END
      SUBROUTINE FCNPEQ (NEQNS, T, Y, P, DYPDP)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NEQNS
      REAL       T, P, Y(NEQNS), DYPDP(NEQNS)
!                                 Define d(DYDT)/dP
      DYPDP(1) = 0.0
      DYPDP(2) = Y(1)**3
      RETURN
      END
      SUBROUTINE FCNPBC (NEQNS, YLEFT, YRIGHT, P, DFDP)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NEQNS
      REAL       P, YLEFT(NEQNS), YRIGHT(NEQNS), DFDP(NEQNS)
!                                 SPECIFICATIONS FOR SUBROUTINES
      EXTERNAL   SSET
!                                 Define dF/dP
      CALL SSET (NEQNS, 0.0, DFDP, 1)
      RETURN
      END
```

Output

```
 I       T              Y1             Y2
 1   0.000000E+00   1.570796E+00  -1.949336E+00
 2   4.444445E-02   1.490495E+00  -1.669567E+00
 3   8.888889E-02   1.421951E+00  -1.419465E+00
 4   1.333333E-01   1.363953E+00  -1.194307E+00
 5   2.000000E-01   1.294526E+00  -8.958461E-01
 6   2.666667E-01   1.243628E+00  -6.373191E-01
 7   3.333334E-01   1.208785E+00  -4.135206E-01
 8   4.000000E-01   1.187783E+00  -2.219351E-01
 9   4.250000E-01   1.183038E+00  -1.584200E-01
10   4.500000E-01   1.179822E+00  -9.973146E-02
11   4.625000E-01   1.178748E+00  -7.233893E-02
12   4.750000E-01   1.178007E+00  -4.638248E-02
13   4.812500E-01   1.177756E+00  -3.399763E-02
14   4.875000E-01   1.177582E+00  -2.205547E-02
15   4.937500E-01   1.177480E+00  -1.061177E-02
16   5.000000E-01   1.177447E+00  -1.479182E-07
17   5.062500E-01   1.177480E+00   1.061153E-02
18   5.125000E-01   1.177582E+00   2.205518E-02
19   5.187500E-01   1.177756E+00   3.399727E-02
20   5.250000E-01   1.178007E+00   4.638219E-02
21   5.375000E-01   1.178748E+00   7.233876E-02
22   5.500000E-01   1.179822E+00   9.973124E-02
23   5.750000E-01   1.183038E+00   1.584199E-01
24   6.000000E-01   1.187783E+00   2.219350E-01
25   6.666667E-01   1.208786E+00   4.135205E-01
26   7.333333E-01   1.243628E+00   6.373190E-01
27   8.000000E-01   1.294526E+00   8.958461E-01
28   8.666667E-01   1.363953E+00   1.194307E+00
29   9.111111E-01   1.421951E+00   1.419465E+00
30   9.555556E-01   1.490495E+00   1.669566E+00
31   1.000000E+00   1.570796E+00   1.949336E+00
Error estimates     3.448358E-06   5.549869E-05
```

BVPMS

Solves a (parameterized) system of differential equations with boundary conditions at two points, using a multiple-shooting method.

Required Arguments

FCNEQN — User-supplied `SUBROUTINE` to evaluate derivatives. The usage is `CALL FCNEQN (NEQNS, T, Y, P, DYDT)`, where

`NEQNS` – Number of equations. (Input)
`T` – Independent variable, *t*. (Input)
`Y` – Array of length `NEQNS` containing the dependent variable. (Input)
`P` – Continuation parameter used in solving highly nonlinear problems. (Input) See Comment 4.
`DYDT` – Array of length `NEQNS` containing *y*′ at `T`. (Output)

The name `FCNEQN` must be declared `EXTERNAL` in the calling program.

FCNJAC — User-supplied `SUBROUTINE` to evaluate the Jacobian. The usage is `CALL FCNJAC (NEQNS, T, Y, P, DYPDY)`, where

NEQNS – Number of equations. (Input)
T – Independent variable. (Input)
Y – Array of length NEQNS containing the dependent variable. (Input)
P – Continuation parameter used in solving highly nonlinear problems. (Input)
See Comment 4.
DYPDY – Array of size NEQNS by NEQNS containing the Jacobian. (Output)
The entry DYPDY(i, j) contains the partial derivative $\partial f_i / \partial y_j$ evaluated at (t, y).

The name FCNJAC must be declared EXTERNAL in the calling program.

FCNBC — User-supplied SUBROUTINE to evaluate the boundary conditions. The usage is CALL FCNBC (NEQNS, YLEFT, YRIGHT, P, H), where

NEQNS – Number of equations. (Input)
YLEFT – Array of length NEQNS containing the values of Y at TLEFT. (Input)
YRIGHT – Array of length NEQNS containing the values of Y at
TRIGHT. (Input)
P – Continuation parameter used in solving highly nonlinear problems. (Input)
See Comment 4.
H – Array of length NEQNS containing the boundary function values. (Output)
The computed solution satisfies (within BTOL) the conditions $h_i = 0$, $i = 1, \ldots,$ NEQNS.

The name FCNBC must be declared EXTERNAL in the calling program.

TLEFT — The left endpoint. (Input)

TRIGHT — The right endpoint. (Input)

NMAX — Maximum number of shooting points to be allowed. (Input)
If NINIT is nonzero, then NMAX must equal NINIT. It must be at least 2.

NFINAL — Number of final shooting points, including the endpoints. (Output)

TFINAL — Vector of length NMAX containing the final shooting points. (Output)
Only the first NFINAL points are significant.

YFINAL — Array of size NEQNS by NMAX containing the values of Y at the points in TFINAL. (Output)

Optional Arguments

NEQNS — Number of differential equations. (Input)

DTOL — Differential equation error tolerance. (Input)
An attempt is made to control the local error in such a way that the global error is proportional to DTOL.
Default: DTOL = 1.0e-4.

BTOL — Boundary condition error tolerance.   (Input)
The computed solution satisfies the boundary conditions, within BTOL tolerance.
Default: BTOL = 1.0e-4.

MAXIT — Maximum number of Newton iterations allowed.   (Input)
Iteration stops if convergence is achieved sooner. Suggested values are MAXIT = 2 for linear problems and MAXIT = 9 for nonlinear problems.
Default: MAXIT = 9.

NINIT — Number of shooting points supplied by the user.   (Input)
It may be 0. A suggested value for the number of shooting points is 10.
Default: NINIT = 0.

TINIT — Vector of length NINIT containing the shooting points supplied by the user. (Input)
If NINIT = 0, then TINIT is not referenced and the routine chooses all of the shooting points. This automatic selection of shooting points may be expensive and should only be used for linear problems. If NINIT is nonzero, then the points must be an increasing sequence with TINIT(1) = TLEFT and TINIT(NINIT) = TRIGHT. By default, TINIT is not used.

YINIT — Array of size NEQNS by NINIT containing an initial guess for the values of Y at the points in TINIT.   (Input)
YINIT is not referenced if NINIT = 0. By default, YINIT is not used.

LDYINI — Leading dimension of YINIT exactly as specified in the dimension statement of the calling program.   (Input)
Default: LDYINI = size (YINIT ,1).

LDYFIN — Leading dimension of YFINAL exactly as specified in the dimension statement of the calling program.   (Input)
Default: LDYFIN = size (YFINAL,1).

FORTRAN 90 Interface

Generic: CALL BVPMS (FCNEQN, FCNJAC, FCNBC, TLEFT, TRIGHT, NMAX, NFINAL, TFINAL, YFINAL [,…])

Specific: The specific interface names are S_BVPMS and D_BVPMS.

FORTRAN 77 Interface

Single: CALL BVPMS (FCNEQN, FCNJAC, FCNBC, NEQNS, TLEFT, TRIGHT, DTOL, BTOL, MAXIT, NINIT, TINIT, YINIT, LDYINI, NMAX, NFINAL, TFINAL, YFINAL, LDYFIN)

Double: The double precision name is DBVPMS.

Example

The differential equations that model an elastic beam are (see Washizu 1968, pages 142−143):

$$\mathbf{M}_{xx} - \frac{\mathbf{NM}}{\mathbf{EI}} + \mathbf{L}(x) = 0$$

$$\mathbf{EIW}_{xx} + \mathbf{M} = 0$$

$$\mathbf{EA}_0\left(\mathbf{U}_x + \mathbf{W}_x^2/2\right) - \mathbf{N} = 0$$

$$\mathbf{N}_x = 0$$

where **U** is the axial displacement, **W** is the transverse displacement, **N** is the axial force, **M** is the bending moment, **E** is the elastic modulus, **I** is the moment of inertia, $\mathbf{A}_0$ is the cross-sectional area, and **L**(x) is the transverse load.

Assume we have a clamped cylindrical beam of radius 0.1in, a length of 10in, and an elastic modulus $\mathbf{E} = 10.6 \times 10^6$ lb/in^2. Then, $\mathbf{I} = 0.784 \times 10^{-4}$, and $\mathbf{A}_0 = \pi 10^{-2}$ in^2, and the boundary conditions are $\mathbf{U} = \mathbf{W} = \mathbf{W}_x = 0$ at each end. If we let $y_1 = \mathbf{U}$, $y_2 = \mathbf{N}/\mathbf{EA}_0$, $y_3 = \mathbf{W}$, $y_4 = \mathbf{W}_x$, $y_5 = \mathbf{M}/\mathbf{EI}$, and $y_6 = \mathbf{M}_x/\mathbf{EI}$, then the above nonlinear equations can be written as a system of six first-order equations.

$$y_1' = y_2 - \frac{y_4^2}{2}$$

$$y_2' = 0$$

$$y_3' = y_4$$

$$y_4' = -y_5$$

$$y_5' = y_6$$

$$y_6' = \frac{\mathbf{A}_0 y_2 y_5}{\mathbf{I}} - \frac{\mathbf{L}(x)}{\mathbf{EI}}$$

The boundary conditions are $y_1 = y_3 = y_4 = 0$ at $x = 0$ and at $x = 10$. The loading function is $\mathbf{L}(x) = -2$, if $3 \le x \le 7$, and is zero elsewhere.

The material parameters, $\mathbf{A}_0$ = `A0`, **I** = `AI`, and **E**, are passed to the evaluation subprograms using the common block `PARAM`.

```
      USE BVPMS_INT
      USE UMACH_INT
      INTEGER    LDY, NEQNS, NMAX
      PARAMETER  (NEQNS=6, NMAX=21, LDY=NEQNS)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    I, MAXIT, NFINAL, NINIT, NOUT
      REAL       TOL, X(NMAX), XLEFT, XRIGHT, Y(LDY,NMAX)
!                                 SPECIFICATIONS FOR COMMON /PARAM/
      COMMON     /PARAM/ A0, A1, E
      REAL       A0, A1, E
!                                 SPECIFICATIONS FOR INTRINSICS
      INTRINSIC  REAL
      REAL       REAL
!                                 SPECIFICATIONS FOR SUBROUTINES
      EXTERNAL   FCNBC, FCNEQN, FCNJAC
```

```
!                                 Set material parameters
      A0 = 3.14E-2
      A1 = 0.784E-4
      E  = 10.6E6
!                                 Set parameters for BVPMS
      XLEFT  = 0.0
      XRIGHT = 10.0
      MAXIT  = 19
      NINIT  = NMAX
      Y = 0.0E0
!                                 Define the shooting points
      DO 10  I=1, NINIT
         X(I) = XLEFT + REAL(I-1)/REAL(NINIT-1)*(XRIGHT-XLEFT)
   10 CONTINUE
!                                 Solve problem
      CALL BVPMS (FCNEQN, FCNJAC, FCNBC, XLEFT, XRIGHT, NMAX, NFINAL, &
                  X, Y,  MAXIT=MAXIT, NINIT=NINIT, TINIT=X, YINIT=Y)

!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,’(26X,A/12X,A,10X,A,7X,A)’) ’Displacement’, &
                                           ’X’, ’Axial’, ’Transvers’// &
                                           ’e’
      WRITE (NOUT,’(F15.1,1P2E15.3)’) (X(I),Y(1,I),Y(3,I),I=1,NFINAL)
      END
      SUBROUTINE FCNEQN (NEQNS, X, Y, P, DYDX)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NEQNS
      REAL       X, P, Y(NEQNS), DYDX(NEQNS)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      REAL       FORCE
!                                 SPECIFICATIONS FOR COMMON /PARAM/
      COMMON     /PARAM/ A0, A1, E
      REAL       A0, A1, E
!                                 Define derivatives
      FORCE = 0.0
      IF (X.GT.3.0 .AND. X.LT.7.0) FORCE = -2.0
      DYDX(1) = Y(2) - P*0.5*Y(4)**2
      DYDX(2) = 0.0
      DYDX(3) = Y(4)
      DYDX(4) = -Y(5)
      DYDX(5) = Y(6)
      DYDX(6) = P*A0*Y(2)*Y(5)/A1 - FORCE/E/A1
      RETURN
      END
      SUBROUTINE FCNBC (NEQNS, YLEFT, YRIGHT, P, F)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NEQNS
      REAL       P, YLEFT(NEQNS), YRIGHT(NEQNS), F(NEQNS)
!                                 SPECIFICATIONS FOR COMMON /PARAM/
      COMMON     /PARAM/ A0, A1, E
      REAL       A0, A1, E
!                                 Define boundary conditions
      F(1) = YLEFT(1)
      F(2) = YLEFT(3)
      F(3) = YLEFT(4)
      F(4) = YRIGHT(1)
      F(5) = YRIGHT(3)
      F(6) = YRIGHT(4)
      RETURN
      END
      SUBROUTINE FCNJAC (NEQNS, X, Y, P, DYPDY)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NEQNS
      REAL       X, P, Y(NEQNS), DYPDY(NEQNS,NEQNS)
!                                 SPECIFICATIONS FOR COMMON /PARAM/
      COMMON     /PARAM/ A0, A1, E
      REAL       A0, A1, E
!                                 SPECIFICATIONS FOR SUBROUTINES
!                                 Define partials, d(DYDX)/dY
      DYPDY = 0.0E0
      DYPDY(1,2) = 1.0
      DYPDY(1,4) = -P*Y(4)
      DYPDY(3,4) = 1.0
      DYPDY(4,5) = -1.0
      DYPDY(5,6) = 1.0
      DYPDY(6,2) = P*Y(5)*A0/A1
      DYPDY(6,5) = P*Y(2)*A0/A1
      RETURN
      END
```

Output

```
                 Displacement
  X          Axial       Transverse
  0.0      1.631E-11     -8.677E-10
  5.0      1.914E-05     -1.273E-03
 10.0      2.839E-05     -4.697E-03
 15.0      2.461E-05     -9.688E-03
 20.0      1.008E-05     -1.567E-02
 25.0     -9.550E-06     -2.206E-02
 30.0     -2.721E-05     -2.830E-02
 35.0     -3.644E-05     -3.382E-02
 40.0     -3.379E-05     -3.811E-02
 45.0     -2.016E-05     -4.083E-02
 50.0     -4.414E-08     -4.176E-02
 55.0      2.006E-05     -4.082E-02
 60.0      3.366E-05     -3.810E-02
 65.0      3.627E-05     -3.380E-02
 70.0      2.702E-05     -2.828E-02
 75.0      9.378E-06     -2.205E-02
 80.0     -1.021E-05     -1.565E-02
 85.0     -2.468E-05     -9.679E-03
 90.0     -2.842E-05     -4.692E-03
 95.0     -1.914E-05     -1.271E-03
100.0      0.000E+00      0.000E+00
```

Comments

1. Workspace may be explicitly provided, if desired, by use of B2PMS/DB2PMS. The reference is:

   ```
   CALL B2PMS (FCNEQN, FCNJAC, FCNBC, NEQNS, TLEFT, TRIGHT, DTOL,
   BTOL, MAXIT, NINIT, TINIT, YINIT, LDYINI, NMAX, NFINAL, TFINAL,
   YFINAL, LDYFIN, WORK, IWK)
   ```

   The additional arguments are as follows:

   WORK — Work array of length `NEQNS * (NEQNS + 1)(NMAX + 12) + NEQNS + 30`.

   IWK — Work array of length `NEQNS`.

2. Informational errors

Type	Code	
1	5	Convergence has been achieved; but to get acceptably accurate approximations to $y(t)$, it is often necessary to start an initial-value solver, for example IVPRK (page 837), at the nearest TFINAL(i) point to t with $t \geq$ TFINAL (i). The vectors YFINAL(j, i), $j = 1, \ldots,$ NEQNS are used as the initial values.
4	1	The initial-value integrator failed. Relax the tolerance DTOL or see Comment 3.
4	2	More than NMAX shooting points are needed for stability.
4	3	Newton's iteration did not converge in MAXIT iterations. If the problem is linear, do an extra iteration. If this error still occurs, check that the routine FCNJAC is giving the correct derivatives. If this does not fix the problem, see Comment 3.
4	4	Linear-equation solver failed. The problem may not have a unique solution, or the problem may be highly nonlinear. In the latter case, see Comment 3.

3. Many linear problems will be successfully solved using program-selected shooting points. Nonlinear problems may require user effort and input data. If the routine fails, then increase NMAX or parameterize the problem. With many shooting points the program essentially uses a finite-difference method, which has less trouble with nonlinearities than shooting methods. After a certain point, however, increasing the number of points will no longer help convergence. To parameterize the problem, see Comment 4.

4. If the problem to be solved is highly nonlinear, then to obtain convergence it may be necessary to embed the problem into a one-parameter family of boundary value problems, $y' = f(t, y, p)$, $h(y(t_a, t_b, p)) = 0$ such that for $p = 0$, the problem is simple, e.g., linear; and for $p = 1$, the stated problem is solved. The routine BVPMS/DBVPMS automatically moves the parameter from $p = 0$ toward $p = 1$.

5. This routine is not recommended for stiff systems of differential equations.

Description

Define N = NEQNS, M = NFINAL, t_a = TLEFT and t_b = TRIGHT. The routine BVPMS uses a multiple-shooting technique to solve the differential equation system $y' = f(t, y)$ with boundary conditions of the form

$$h_k(y_1(t_a), \ldots, y_N(t_a), y_1(t_b), \ldots, y_N(t_b)) = 0 \quad \text{for } k = 1, \ldots, N$$

A modified version of IVPRK, page 837, is used to compute the initial-value problem at each "shot." If there are M shooting points (including the endpoints t_a and t_b), then a system of NM simultaneous nonlinear equations must be solved. Newton's method is used to solve this system, which has a Jacobian matrix with a "periodic band" structure. Evaluation of the NM functions and the $NM \times NM$ (almost banded) Jacobian for one iteration of Newton's method is accomplished in one pass from t_a to t_b of the modified IVPRK, operating on a system of $N(N + 1)$ differential equations. For most problems, the total amount of work should not be highly dependent on M. Multiple shooting avoids many of the serious ill-conditioning problems that plague simple shooting methods. For more details on the algorithm, see Sewell (1982).

The boundary functions should be scaled so that all components h_k are of comparable magnitude since the absolute error in each is controlled.

DASPG

Solves a first order differential-algebraic system of equations, $g(t, y, y') = 0$, using the Petzold–Gear BDF method.

Required Arguments

T — Independent variable, t. (Input/Output)
Set T to the starting value t_0 at the first step.

TOUT — Final value of the independent variable. (Input)
Update this value when re-entering after output, IDO = 2.

IDO — Flag indicating the state of the computation. (Input/Output)

IDO	State
1	Initial entry
2	Normal re-entry after obtaining output
3	Release workspace
4	Return because of an error condition

The user sets IDO = 1 or IDO = 3. All other values of IDO are defined as output. The initial call is made with IDO = 1 and T = t_0. The routine then sets IDO = 2, and this value is used for all but the last entry that is made with IDO = 3. This call is used to release workspace and other final tasks. Values of IDO larger than 4 occur only when calling the second-level routine D2SPG and using the options associated with reverse communication.

Y — Array of size NEQ containing the dependent variable values, y. This array must contain initial values. (Input/Output)

YPR — Array of size NEQ containing derivative values, y'. This array must contain initial values. (Input/Output)
The routine will solve for consistent values of y' to satisfy the equations at the starting point.

GCN — User-supplied SUBROUTINE to evaluate $g(t, y, y')$. The usage is CALL GCN (NEQ, T, Y, YPR, GVAL), where GCN must be declared EXTERNAL in the calling program. The routine will solve for values of $y'(t_0)$ so that $g(t_0, y, y') = 0$. The user can signal that g is not defined at requested values of (t, y, y') using an option. This causes the routine to reduce the step size or else quit.

NEQ – Number of differential equations. (Input)
T – Independent variable. (Input)
Y – Array of size NEQ containing the dependent variable values $y(t)$. (Input)
YPR – Array of size NEQ containing the derivative values $y'(t)$. (Input)
GVAL – Array of size NEQ containing the function values, $g(t, y, y')$. (Output)

Optional Arguments

NEQ — Number of differential equations. (Input)
Default: NEQ = size(y,1)

FORTRAN 90 Interface

Generic: CALL DASPG (T, TOUT, IDO, Y, YPR, GCN[,…])

Specific: The specific interface names are S_DASPG and D_DASPG.

FORTRAN 77 Interface

Single: CALL DASPG (NEQ, T, TOUT, IDO, Y, YPR, GCN)

Double: The double precision name is DDASPG.

Example 1

The Van der Pol equation $u'' + \mu(u^2 - 1)\,u' + u = 0$, $\mu > 0$, is a single ordinary differential equation with a periodic limit cycle. See Hartman (1964, page 181). For the value $\mu = 5$, the equations are integrated from $t = 0$ until the limit has clearly developed at $t = 26$. The (arbitrary) initial conditions used here are $u(0) = 2$ and $u'(0) = -\,2/3$. Except for these initial conditions and the final t value, this is problem (E2) of the Enright and Pryce (1987) test package. This equation is solved as a differential-algebraic system by defining the first-order system:

$$
\begin{aligned}
\varepsilon &= 1/\mu \\
y_1 &= u \\
g_1 &= y_2 - y_1' = 0 \\
g_2 &= \left(1 - y_1^2\right) y_2 - \varepsilon\left(y_1 + y_2'\right) = 0
\end{aligned}
$$

Note that the initial condition for

$$y_2'$$

in the sample program is not consistent, $g_2 \neq 0$ at $t = 0$. The routine DASPG solves for this starting value. No options need to be changed for this usage. The set of pairs $(u(t_j), u'(t_j))$ are accumulated for the 260 values $t_j = 0.1, 26, (0.1)$.

```
      USE UMACH_INT
      USE DASPG_INT
      INTEGER    N, NP
      PARAMETER  (N=2, NP=260)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    ISTEP, NOUT, NSTEP
      REAL DELT,  T, TEND, U(NP), UPR(NP), Y(N), YPR(N)
!                                 SPECIFICATIONS FOR FUNCTIONS
      EXTERNAL   GCN
!                                 Define initial data
      IDO = 1
      T = 0.0
      TEND = 26.0
      DELT = 0.1
      NSTEP = TEND/DELT
!                                 Initial values
      Y(1) = 2.0
      Y(2) = -2.0/3.0
!                                 Initial derivatives
      YPR(1) = Y(2)
      YPR(2) = 0.
!                                 Write title
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99998)
!                                 Integrate ODE/DAE
      ISTEP = 0
   10 CONTINUE
      ISTEP = ISTEP + 1
      CALL DASPG (T, T+DELT, IDO, Y, YPR, GCN)
```

```
!                                 Save solution for plotting
      IF (ISTEP .LE. NSTEP) THEN
         U(ISTEP) = Y(1)
         UPR(ISTEP) = YPR(1)
!                                 Release work space
         IF (ISTEP .EQ. NSTEP) IDO = 3
         GO TO 10
      END IF
      WRITE (NOUT,99999) TEND, Y, YPR
99998 FORMAT (11X, ’T’, 14X, ’Y(1)’, 11X, ’Y(2)’, 10X, ’Y’’(1)’, 10X, &
            ’Y’’(2)’)
99999 FORMAT (5F15.5)
!                                 Start plotting
!      CALL SCATR (NSTEP, U, UPR)
!      CALL EFSPLT (0, ’ ’)
      END
!
      SUBROUTINE GCN (N, T, Y, YPR, GVAL)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    N
      REAL T, Y(N), YPR(N), GVAL(N)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      REAL EPS
!
      EPS = 0.2
!
      GVAL(1) = Y(2) - YPR(1)
      GVAL(2) = (1.0-Y(1)**2)*Y(2) - EPS*(Y(1)+YPR(2))
      RETURN
      END
```

Output

```
    T              Y(1)           Y(2)          Y’(1)          Y’(2)
26.00000        1.45330       -0.24486       -0.24713       -0.09399
```

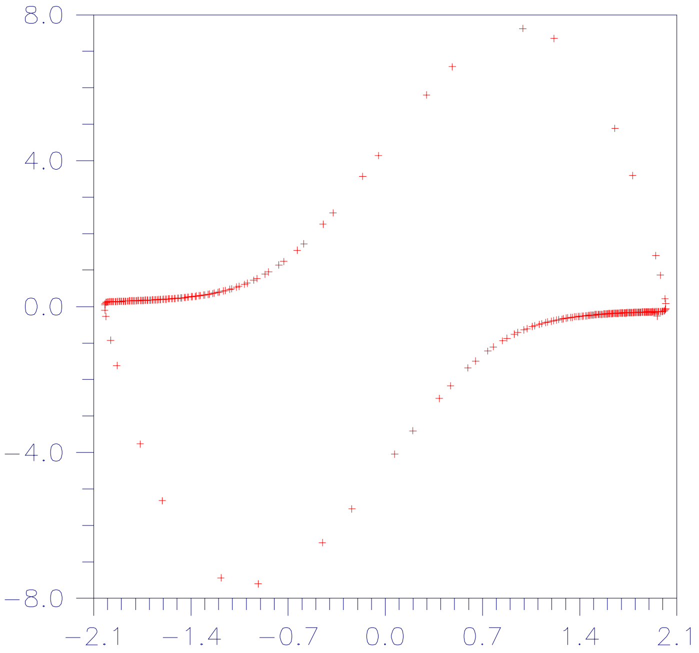


Figure 5-1 Van der Pol Cycle, ($u(t)$, $u'(t)$), $\mu = 5$.

Comments

Users can often get started using the routine DASPG/DDASPG without reading beyond this point in the documentation. There is often no reason to use options when getting started. Those readers who do not want to use options can turn directly to the first two examples. The following tables give numbers and key phrases for the options. A detailed guide to the options is given below in Comment 2.

Value	Brief or Key Phrase for INTEGER Option
6	INTEGER option numbers
7	Floating-point option numbers
IN(1)	First call to DASPG, D2SPG
IN(2)	Scalar or vector tolerances
IN(3)	Return for output at intermediate steps
IN(4)	Creep up on special point, TSTOP
IN(5)	Provide (analytic) partial derivative formulas
IN(6)	Maximum number of steps
IN(7)	Control maximum step size
IN(8)	Control initial step size

Value	Brief or Key Phrase for INTEGER Option
IN(9)	*Not Used*
IN(10)	Constrain dependent variables
IN(11)	Consistent initial data
IN(12-15)	*Not Used*
IN(16)	Number of equations
IN(17)	What routine did, if any errors
IN(18)	Maximum BDF order
IN(19)	Order of BDF on next move
IN(20)	Order of BDF on previous move
IN(21)	Number of steps
IN(22)	Number of *g* evaluations
IN(23)	Number of derivative matrix evaluations
IN(24)	Number of error test failures
IN(25)	Number of convergence test failures
IN(26)	Reverse communiction for *g*
IN(27)	Where is *g* stored?
IN(28)	Panic flag
IN(29)	Reverse communication, for partials
IN(30)	Where are partials stored?
IN(31)	Reverse communication, for solving
IN(32)	*Not Used*
IN(33)	Where are vector tolerances stored?
IN(34)	Is partial derivative array allocated?
IN(35)	User's work arrays sizes are checked
IN(36-50)	*Not used*

Table 1. Key Phrases for Floating-Point Options

Value	Brief or Key Phrase for Floating-Point Option
INR(1)	Value of *t*
INR(2)	Farthest internal *t* vaue of integration
INR(3)	Value of `TOUT`
INR(4)	A stopping point of integration before `TOUT`
INR(5)	Values of two scalars `ATOL`, `RTOL`
INR(6)	Initial step size to use
INR(7)	Maximum step allowed
INR(8)	Condition number reciprocal
INR(9)	Value of c_j for partials
INR(10)	Step size on the next move
INR(11)	Step size on the previous move
INR(12-20)	*Not Used*

Table 2. Number and Key Phrases for Floating-Point Options

1. Workspace may be explicitly provided, and many of the options utilized by directly calling `D2SPG/DD2SPG`. The reference is:

```
CALL D2SPG (N, T, TOUT, IDO, Y, YPR, GCN, JGCN, IWK, WK)
```

The additional arguments are as follows:

IDO	**State**
5	Return for evaluation of $g(t, y, y')$
6	Return for evaluation of matrix $A = [\partial g/\partial y + c_j \partial g/\partial y']$
7	Return for factorization of the matrix $A = [\partial g/\partial y + c_j \partial g/\partial y']$
8	Return for solution of $A\Delta y = \Delta g$

These values of `IDO` occur only when calling the second-level routine `D2SPG` and using options associated with reverse communication. The routine `D2SPG/DD2SPG` is reentered.

GCN — A Fortran SUBROUTINE to compute $g(t, y, y')$. This routine is normally provided by the user. That is the default case. The dummy IMSL routine `DGSPG/DDGSPG` may be used as this argument when $g(t, y, y')$ is evaluated by reverse communication. In either case, a name must be declared in a Fortran `EXTERNAL` statement. If usage of the dummy IMSL routine is intended, then the name `DGSPG/DDGSPG` should be specified. The dummy IMSL routine will never be called under this optional usage of reverse communication. An example of reverse communication for evaluation of g is given in Example 4.

JGCN — A Fortran SUBROUTINE to compute partial derivatives of $g(t, y, y')$. This routine may be provided by the user. The dummy IMSL routine `DJSPG/DDJSPG` may be used as this argument when partial derivatives are computed using divided differences. This is the default. The dummy routine is not called under default conditions. If partial derivatives are to be explicitly provided, the routine JGCN must be written by the user or reverse communication can be used. An example of reverse communication for evaluation of the partials is given in Example 4.

If the user writes a routine with the *fixed* name `DJSPG/DDJSPG`, then partial derivatives can be provided while calling `DASPG`. An option is used to signal that formulas for partial derivatives are being supplied. This is illustrated in Example 3. The name of the partial derivative routine must be declared in a Fortran `EXTERNAL` statement when calling `D2SPG`. If usage of the dummy IMSL routine is intended, then the name `DJSPG/DDJSPG` should be specified for this `EXTERNAL` name. Whenever the user provides partial derivative evaluation formulas, by whatever means, that must be noted with an option. Usage of the derivative evaluation routine is `CALL JGCN (N, T, Y, YPR, CJ, PDG, LDPDG)` where

Arg **Definition**

`N` Number of equations. (Input)

`T` Independent variable, t. (Input)

`Y` Array of size `N` containing the values of the dependent variables, y. (Input)

`YPR` Array of size `N` containing the values of the derivatives, y'. (Input)

`CJ` The value c_j used in computing the partial derivatives returned in `PDG`. (Input)

`PDG` Array of size `LDPDG * N` containing the partial derivatives $A = [\partial g/\partial y + c_j \partial g/\partial y']$. Each nonzero derivative entry a_{ij} is returned in the array location `PDG(i, j)`. The array contents are zero when the routine is called. Thus, only the nonzero derivatives have to be defined in the routine `JGCN`. (Output)

`LDPDG` The leading dimension of `PDG`. Normally, this value is `N`. It is a value larger than `N` under the conditions explained in option **16** of `LSLRG` (Chapter 1, Linear Systems).

`JGCN` must be declared `EXTERNAL` in the calling program.

IWK — Work array of integer values. The size of this array is 35 + N. The contents of IWK must not be changed from the first call with IDO = 1 until after the final call with
IDO = 3.

WK — Work ahrray of floating-point values in the working precision. The size of this array is 41 + (MAXORD + 6)N + (N + K)N(1 − L) where K is determined from the values IVAL(3) and IVAL(4) of option **16** of LSLRG (Chapter 1, Linear Systems). The value of L is 0 unless option **IN(34)** is used to avoid allocation of the array containing the partial derivatives. With the use of this option, L can be set to 1. The contents of array WK must not be changed from the first call with IDO = 1 until after the final call.

2. Integer and Floating-Point Options with Chapter 11 Options Manager

The routine DASPG allows the user access to many interface parameters and internal working variables by the use of options. The options manager subprograms IUMAG, SUMAG, and DUMAG (Chapter 11, Utilities), are used to change options from their default values or obtain the current values of required parameters.

Options of type INTEGER:

6 This is the list of numbers used for INTEGER options. Users will typically call this option first to get the numbers, IN(I), I = 1, 50. This option has 50 entries. The default values are IN(I) = I + 50, I = 1, 50.

7 This is the list of numbers used for REAL and DOUBLE PRECISION options. Users will typically call this option first to get the numbers, INR(I), I = 1,20. This option has 20 entries. The default values are INR(I) = I + 50, I = 1, 20.

IN(1) This is the first call to the routine DASPG or D2SPG. Value is 0 for the first call, 1 for further calls. Setting IDO = 1 resets this option to its default. Default value is 0.

IN(2) This flag controls the kind of tolerances to be used for the solution. Value is 0 for scalar values of absolute and relative tolerances applied to all components. Value is 1 when arrays for both these quantities are specified. In this case, the option **IN(33)** is used to get the offset into WK where the 2N array values are to be placed: all ATOL values followed by all RTOL values. This offset is defined after the call to the routine D2SPG so users will have to call the options manager at a convenient place in the GCN routine or during reverse communication. Default value is 0.

IN(3) This flag controls when the code returns to the user with output values of *y* and *y* ′. If the value is 0, it returns to the user at T = TOUT only. If the value is 1, it returns to the user at an internal working step. Default value is 0.

IN(4) This flag controls whether the code should integrate past a special point, `TSTOP`, and then interpolate to get y and y'at `TOUT`. If the value is 0, this is permitted. If the value is 1, the code assumes the equations either change on the alternate side of `TSTOP` or they are undefined there. In this case, the code creeps up to `TSTOP` in the direction of integration. The value of `TSTOP` is set with option `INR`(4). Default value is 0.

IN(5) This flag controls whether partial derivatives are computed using divided onesided differences, or they are to be computed using user-supplied evaluation formulas. If the value is 0, use divided differences. If the value is 1, use formulas for the partial derivatives. See Example 3 for an illustration of one way to do this. Default value is 0.

IN(6) The maximum number of steps. Default value is 500.

IN(7) This flag controls a maximum magnitude constraint for the step size. If the value is 0, the routine picks its own maximum. If the value is 1, a maximum is specified by the user. That value is set with option number **INR(7)**. Default value is 0.

IN(8) This flag controls an initial value for the step size. If the value is 0, the routine picks its own initial step size. If the value is 1, a starting step size is specified by the user. That value is set with option number **INR(6)**. Default value is 0.

IN(9) Not used. Default value is 0.

IN(10) This flag controls attempts to constrain all components to be nonnegative. If the value is 0, no constraints are enforced. If value is 1, constraint is enforced. Default value is 0.

IN(11) This flag controls whether the initial values (t, y, y') are consistent. If the value is 0, $g(t, y, y') = 0$ at the initial point. If the value is 1, the routine will try to solve for y' to make this equation satisfied. Default value is 1.

IN(12-15) Not used. Default value is 0 for each option.

IN(16) The number of equations in the system, n. Default value is 0.

IN(17) This value reports what the routine did. Default value is 0.

Value	Explanation
1	A step was taken in the intermediate output mode. The value `TOUT` has not been reached.
2	The integration to exactly `TSTOP` was completed.
3	The integration to `TSTOP` was completed by stepping past `TSTOP` and interpolating to evaluate y and y'.
−1	Too many steps taken.
−2	Error tolerances are too small.
−3	A pure relative error tolerance can't be satisfied.
−6	There were repeated error test failures on the last step.
−7	The BDF corrector equation solver did not converge.
−8	The matrix of partial derivatives is singular.
−10	The BDF corrector equation solver did not converge because the evaluation failure flag was raised.
−11	The evaluation failure flag was raised to quit.
−12	The iteration for the initial vaule of y' did not converge.
−33	There is a fatal error, perhaps caused by invalid input.

Table 3. What the Routine `DASPG` or `D2SPG` Did

IN(18) The maximum order of BDF formula the routine should use. Default value is 5.

IN(19) The order of the BDF method the routine will use on the next step. Default value is `IMACH`(5).

IN(20) The order of the BDF method used on the last step. Default value is `IMACH`(5).

IN(21) The number of steps taken so far. Default value is 0.

IN(22) The number of times that g has been evaluated. Default value is 0.

IN(23) The number of times that the partial derivative matrix has been evaluated. Default value is 0.

IN(24) The total number of error test failures so far. Default value is 0.

IN(25) The total number of convergence test failures so far. This includes singular iteration matrices. Default value is 0.

IN(26) Use reverse communication to evaluate g when this value is 0. If the value is 1, forward communication is used. Use the routine `D2SPG` for reverse communication. With reverse communication, a return will be made with `IDO` = 5. Compute the value of *g*, place it into the array `WK` at the offset obtained with option **IN(27**), and re-enter the routine. Default value is 1.

IN(27) The user is to store the evaluated function *g* during reverse communication in the work array `WK` using this value as an offset. Default value is `IMACH`(5).

IN(28) This value is a "panic flag." After an evaluation of *g*, this value is checked. The value of *g* is used if the flag is 0. If it has the value −1, the routine reduces the step size and possibly the order of the BDF. If the value is −2, the routine returns control to the user immediately. This option is also used to signal a singular or poorly conditioned partial derivative matrix encountered during the factor phase in reverse communication. Use a nonzero value when the matrix is singular. Default value is 0.

IN(29) Use reverse communication to evaluate the partial derivative matrix when this value is 0. If the value is 1, forward communication is used. Use the routine `D2SPG` for reverse communication. With reverse communication, a return will be made with `IDO` = 6. Compute the partial derivative matrix *A* and re-enter the routine. If forward communication is used for the linear solver, return the partials using the offset into the array `WK`. This offset value is obtained with option **IN(30)**. Default value is 1.

IN(30) The user is to store the values of the partial derivative matrix *A* by columns in the work array `WK` using this value as an offset. The option **16** for `LSLRG` is used here to compute the row dimension of the internal working array that contains *A*. Users can also choose to store this matrix in some convenient form in their calling program if they are providing linear system solving using reverse communication. See options **IN(31)** and **IN(34)**. Default value is `IMACH`(5).

IN(31) Use reverse communication to solve the linear system $A\Delta y = \Delta g$ if this value is 0. If the value is 1, use forward communication into the routines `L2CRG` and `LFSRG` (Chapter 1, Linear Systems) for the linear system solving. Return the solution using the offset into the array `WK` where *g* is stored. This offset value is obtained with option **IN(27)**. With reverse communication, a return will be made with `IDO` = 7 for factorization of *A* and with `IDO` = 8 for solving the system. Re-enter the routine in both cases. If the matrix A is singular or poorly conditioned, raise the "panic flag," option **IN(28)**, during the factorization. Default value is 1.

IN(32) Not used. Default value is 0.

IN(33) The user is to store the vector of values for `ATOL` and `RTOL` in the array `WK` using this value as an offset. The routine `D2SPG` must be called before this value is defined.

IN(34) This flag is used if the user has not allocated storage for the matrix *A* in the array `WK`. If the value is 0, storage is allocated. If the value is 1, storage was not allocated. In this case, the user must be using reverse communication to evaluate the partial derivative matrix and to solve the linear systems $A\Delta y = \Delta g$. Default value is 0.

IN(35) These two values are the sizes of the arrays IWK and WK allocated in the users program. The values are checked against the program requirements. These checks are made only if the values are positive. Users will normally set this option when directly calling D2SPG. Default values are (0, 0).

Options of type REAL or DOUBLE PRECISION:

INR(1) The value of the independent variable, *t*. Default value is AMACH(6).

INR(2) The farthest working *t* point the integration has reached. Default value is AMACH(6) .

INR(3) The current value of TOUT. Default value is AMACH(6).

INR(4) The next special point, TSTOP, before reaching TOUT. Default value is AMACH(6). Used with option **IN(4)**.

INR(5) The pair of scalar values ATOL and RTOL that apply to the error estimates of all components of *y*. Default values for both are SQRT(AMACH(4)).

INR(6) The initial step size if DASPG is not to compute it internally. Default value is AMACH(6).

INR(7) The maximum step size allowed. Default value is AMACH(2).

INR(8) This value is the reciprocal of the condition number of the matrix *A*. It is defined when forward communication is used to solve for the linear updates to the BDF corrector equation. No further program action, such as declaring a singular system, based on the condition number. Users can declare the system to be singular by raising the "panic flag" using option **IN(28)**. Default value is AMACH(6).

INR(9) The value of c_j used in the partial derivative matrix for reverse communication evaluation. Default value is AMACH(6).

INR(10) The step size to be attempted on the next move. Default value is AMACH(6).

INR(11) The step size taken on the previous move. Default value is AMACH(6).

4. Norm Function Subprogram

The routine DASPG uses a weighted Euclidean-RMS norm to measure the size of the estimated error in each step. This is done using a FUNCTION subprogram: REAL

FUNCTION D10PG (N, V, WT). This routine returns the value of the RMS weighted norm given by:

$$\text{D10PG} = \sqrt{N^{-1}\sum_{i=1}^{N}\left(v_i / wt_i\right)^2}$$

Users can replace this function with one of their own choice. This should be done only for problem-related reasons.

Description

Routine DASPG finds an approximation to the solution of a system of differential-algebraic equations $g(t, y, y') = 0$, with given initial data for y and y'. The routine uses BDF formulas, appropriate for systems of stiff ODEs, and attempts to keep the global error proportional to a user-specified tolerance. See Brenan et al. (1989). This routine is efficient for stiff systems of index 1 or index 0. See Brenan et al. (1989) for a definition of *index*. Users are encouraged to use DOUBLE PRECISION accuracy on machines with a short REAL precision accuracy. The examples given below are in REAL accuracy because of the desire for consistency with the rest of IMSL MATH/LIBRARY examples. The routine DASPG is based on the code DASSL designed by L. Petzold (1982-1990).

Example 2

The first-order equations of motion of a point-mass m suspended on a massless wire of length ℓ under the influence of gravity force, mg and tension value λ, in Cartesian coordinates, (p, q), are

$$\begin{aligned} p' &= u \\ q' &= v \\ mu' &= -p\lambda \\ mv' &= -q\lambda - mg \\ p^2 + q^2 - \ell^2 &= 0 \end{aligned}$$

This is a genuine differential-algebraic system. The problem, as stated, has an index number equal to the value 3. Thus, it cannot be solved with DASPG directly. Unfortunately, the fact that the index is greater than 1 must be deduced indirectly. Typically there will be an error processed which states that the (BDF) corrector equation did not converge. The user then differentiates and replaces the constraint equation. This example is transformed to a problem of index number of value 1 by differentiating the last equation twice. This resulting equation, which replaces the given equation, is the total energy balance:

$$m(u^2 + v^2) - mgq - \ell^2\lambda = 0$$

With initial conditions and systematic definitions of the dependent variables, the system becomes:

$$p(0) = \ell, q(0) = u(0) = v(0) = \lambda(0) = 0$$

$$\begin{aligned} y_1 &= p \\ y_2 &= q \\ y_3 &= u \\ y_4 &= v \\ y_5 &= \lambda \end{aligned}$$

$$\begin{aligned} g_1 &= y_3 - y_1' = 0 \\ g_2 &= y_4 - y_2' = 0 \\ g_3 &= -y_1 y_5 - m y_3' = 0 \\ g_4 &= -y_2 y_5 - mg - m y_4' = 0 \\ g_5 &= m\left(y_3^2 + y_4^2\right) - mg y_2 - \ell^2 y_5 = 0 \end{aligned}$$

The problem is given in English measurement units of feet, pounds, and seconds. The wire has length 6.5 *ft*, and the mass at the end is 98 *lb*. Usage of the software does not require it, but standard or "SI" units are used in the numerical model. This conversion of units is done as a first step in the user-supplied evaluation routine, GCN. A set of initial conditions, corresponding to the pendulum starting in a horizontal position, are provided as output for the input signal of n = 0. The maximum magnitude of the tension parameter, $\lambda(t) = y_5(t)$, is computed at the output points, t = 0.1, π, (0.1). This extreme value is converted to English units and printed.

```
      USE DASPG_INT
      USE CUNIT_INT
      USE UMACH_INT
      USE CONST_INT
      INTEGER    N
      PARAMETER  (N=5)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    IDO, ISTEP, NOUT, NSTEP
      REAL       DELT, GVAL(N), MAXLB, MAXTEN, T, TEND, TMAX, Y(N), &
                YPR(N)
!                                 SPECIFICATIONS FOR INTRINSICS
      INTRINSIC  ABS
      REAL       ABS
!                                 SPECIFICATIONS FOR SUBROUTINES
      EXTERNAL   GCN
!                                 SPECIFICATIONS FOR FUNCTIONS
!                                 Define initial data
      IDO   = 1
      T     = 0.0
      TEND  = CONST('pi')
      DELT  = 0.1
      NSTEP = TEND/DELT
      CALL UMACH (2, NOUT)
!                                 Get initial conditions
      CALL GCN (0, T, Y, YPR, GVAL)
      ISTEP  = 0
      MAXTEN = 0.
   10 CONTINUE
      ISTEP = ISTEP + 1
      CALL DASPG (T, T+DELT, IDO, Y, YPR, GCN)
      IF (ISTEP .LE. NSTEP) THEN
!                                 Note max tension value
         IF (ABS(Y(5)) .GT. ABS(MAXTEN)) THEN
            TMAX   = T
            MAXTEN = Y(5)
         END IF
         IF (ISTEP .EQ. NSTEP) IDO = 3
         GO TO 10
      END IF
!                                  Convert to English units
      CALL CUNIT (MAXTEN, ’kg/s**2’, MAXLB, ’lb/s**2’)
!                                  Print maximum tension
      WRITE (NOUT,99999) MAXLB, TMAX
99999 FORMAT (’ Extreme string tension of’, F10.2, ’ (lb/s**2)’, &
            ’ occurred at ’, ’time ’, F10.2)
      END
!
      SUBROUTINE GCN (N, T, Y, YPR, GVAL)
      USE CUNIT_INT
      USE CONST_INT
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    N
      REAL       T, Y(*), YPR(*), GVAL(*)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      REAL       FEETL, GRAV, LENSQ, MASSKG, MASSLB, METERL, MG
!                                 SPECIFICATIONS FOR SAVE VARIABLES
      LOGICAL    FIRST
      SAVE       FIRST
!                                 SPECIFICATIONS FOR SUBROUTINES
!                                 SPECIFICATIONS FOR FUNCTIONS
!
      DATA FIRST/.TRUE./
!
      IF (FIRST) GO TO 20
   10 CONTINUE
!                                Define initial conditions
      IF (N .EQ. 0) THEN
!                                The pendulum is horizontal
!                                with these initial y values
         Y(1)   = METERL
         Y(2)   = 0.
         Y(3)   = 0.
         Y(4)   = 0.
         Y(5)   = 0.
         YPR(1) = 0.
         YPR(2) = 0.
         YPR(3) = 0.
         YPR(4) = 0.
         YPR(5) = 0.
         RETURN
      END IF
!                                Compute residuals
      GVAL(1) = Y(3) - YPR(1)
      GVAL(2) = Y(4) - YPR(2)
      GVAL(3) = -Y(1)*Y(5) - MASSKG*YPR(3)
      GVAL(4) = -Y(2)*Y(5) - MASSKG*YPR(4) - MG
      GVAL(5) = MASSKG*(Y(3)**2+Y(4)**2) - MG*Y(2) - LENSQ*Y(5)
      RETURN
!                                Convert from English to
!                                Metric units:
   20 CONTINUE
      FEETL  = 6.5
      MASSLB = 98.0
!                                Change to meters
      CALL CUNIT (FEETL, ’ft’, METERL, ’meter’)
!                                Change to kilograms
      CALL CUNIT (MASSLB, ’lb’, MASSKG, ’kg’)
!                                Get standard gravity
      GRAV  = CONST(’StandardGravity’)
      MG    = MASSKG*GRAV
      LENSQ = METERL**2
      FIRST = .FALSE.
      GO TO 10
      END
```

Output

```
Extreme string tension of   1457.24 (lb/s**2) occurred at time       2.50
```

Example 3

In this example, we solve a stiff ordinary differential equation (E5) from the test package of Enright and Pryce (1987). The problem is nonlinear with nonreal eigenvalues. It is included as an example because it is a stiff problem, and its partial derivatives are provided in the usersupplied routine with the fixed name DJSPG. Users who require a variable routine name for partial derivatives can use the routine D2SPG. Providing explicit formulas for partial derivatives is an important consideration for problems where evaluations of the function $g(t, y, y')$ are expensive. Signaling that a derivative matrix is provided requires a call to the Chapter 10 options manager utility, IUMAG. In addition, an initial integration step-size is given for this test problem. A signal for this is passed using the options manager routine IUMAG. The error tolerance is changed from the defaults to a pure absolute tolerance of 0.1 * SQRT(AMACH(4)). Also see IUMAG, SUMAG and DUMAG in Chapter 11, Utilities, for further details about the options manager routines.

```
      USE IMSL_LIBRARIES
      INTEGER    N
      PARAMETER  (N=4)
!                                 SPECIFICATIONS FOR PARAMETERS
      INTEGER    ICHAP, IGET, INUM, IPUT, IRNUM
      PARAMETER  (ICHAP=5, IGET=1, INUM=6, IPUT=2, IRNUM=7)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    IDO, IN(50), INR(20), IOPT(2), IVAL(2), NOUT
      REAL       C0, PREC, SVAL(3), T, TEND, Y(N), YPR(N)
!                                 SPECIFICATIONS FOR FUNCTIONS
```

```
      EXTERNAL   GCN
!                                 Define initial data
      IDO = 1
      T = 0.0
      TEND = 1000.0
!                                 Initial values
      C0 = 1.76E-3
      Y(1) = C0
      Y(2) = 0.
      Y(3) = 0.
      Y(4) = 0.
!                                 Initial derivatives
      YPR(1) = 0.
      YPR(2) = 0.
      YPR(3) = 0.
      YPR(4) = 0.
!                                 Get option numbers
      IOPT(1) = INUM
      CALL IUMAG ('math', ICHAP, IGET, 1, IOPT, IN)
      IOPT(1) = IRNUM
      CALL IUMAG ('math', ICHAP, IGET, 1, IOPT, INR)
!                                 Provide initial step
      IOPT(1) = INR(6)
      SVAL(1) = 5.0E-5
!                                 Provide absolute tolerance
      IOPT(2) = INR(5)
      PREC = AMACH(4)
      SVAL(2) = 0.1*SQRT(PREC)
      SVAL(3) = 0.0

      CALL UMAG ('math', ICHAP, IPUT, IOPT, SVAL)
!                                 Using derivatives and
      IOPT(1) = IN(5)
      IVAL(1) = 1
!                                 providing initial step
      IOPT(2) = IN(8)
      IVAL(2) = 1

      CALL IUMAG ('math', ICHAP, IPUT, 2, IOPT, IVAL)
!                                 Write title
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99998)
!                                 Integrate ODE/DAE
      CALL DASPG (T, TEND, IDO, Y, YPR, GCN)
      WRITE (NOUT,99999) T, Y, YPR
!                                 Reset floating options
!                                 to defaults
      IOPT(1) = -INR(5)
      IOPT(2) = -INR(6)
!
      CALL UMAG ('math', ICHAP, IPUT, IOPT, SVAL)
!                                 Reset integer options
!                                 to defaults
      IOPT(1) = -IN(5)
      IOPT(2) = -IN(8)
```

```
!
      CALL IUMAG (’math’, ICHAP, IPUT, 2, IOPT, IVAL)

99998 FORMAT (11X, ’T’, 14X, ’Y followed by Y’’’)
99999 FORMAT (F15.5/(4F15.5))
      END
!
      SUBROUTINE GCN (N, T, Y, YPR, GVAL)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    N
      REAL       T, Y(N), YPR(N), GVAL(N)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      REAL       C1, C2, C3, C4
!
      C1 = 7.89E-10
      C2 = 1.1E7
      C3 = 1.13E9
      C4 = 1.13E3
!
      GVAL(1) = -C1*Y(1) - C2*Y(1)*Y(3) - YPR(1)
      GVAL(2) = C1*Y(1) - C3*Y(2)*Y(3) - YPR(2)
      GVAL(3) = C1*Y(1) - C2*Y(1)*Y(3) + C4*Y(4) - C3*Y(2)*Y(3) - &
               YPR(3)
      GVAL(4) = C2*Y(1)*Y(3) - C4*Y(4) - YPR(4)
      RETURN
      END
      SUBROUTINE DJSPG (N, T, Y, YPR, CJ, PDG, LDPDG)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    N, LDPDG
      REAL       T, CJ, Y(N), YPR(N), PDG(LDPDG,N)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      REAL       C1, C2, C3, C4
!
      C1 = 7.89E-10
      C2 = 1.1E7
      C3 = 1.13E9
      C4 = 1.13E3
!
      PDG(1,1) = -C1 - C2*Y(3) - CJ
      PDG(1,3) = -C2*Y(1)
      PDG(2,1) = C1
      PDG(2,2) = -C3*Y(3) - CJ
      PDG(2,3) = -C3*Y(2)
      PDG(3,1) = C1 - C2*Y(3)
      PDG(3,2) = -C3*Y(3)
      PDG(3,3) = -C2*Y(1) - C3*Y(2) - CJ
      PDG(3,4) = C4
      PDG(4,1) = C2*Y(3)
      PDG(4,3) = C2*Y(1)
      PDG(4,4) = -C4 - CJ
      RETURN
      END
```

Output

```
     T              Y followed by Y'
1000.00000
   0.00162        0.00000        0.00000        0.00000
   0.00000        0.00000        0.00000        0.00000
```

Example 4

In this final example, we compute the solution of $n = 10$ ordinary differential equations, $g = Hy - y'$, where $y(0) = y_0 = (1, 1, \ldots, 1)^T$. The value

$$\sum_{i=1}^{n} y_i\left(t\right)$$

is evaluated at $t = 1$. The constant matrix H has entries $h_{i,j} = \min(j - i, 0)$ so it is lower Hessenberg. We use reverse communication for the evaluation of the following intermediate quantities:

1. The function g,
2. The partial derivative matrix $A = \partial g/\partial y + c_j \partial g/\partial y' = H - c_j I$,
3. The solution of the linear system $A\Delta y = \Delta g$.

In addition to the use of reverse communication, we evaluate the partial derivatives using formulas. No storage is allocated in the floating-point work array for the matrix. Instead, the matrix A is stored in an array A within the main program unit. Signals for this organization are passed using the routine IUMAG (Chapter 11, Utilities).

An algorithm appropriate for this matrix, Givens transformations applied from the right side, is used to factor the matrix A. The rotations are reconstructed during the solve step. See SROTG (Chapter 9, Basic Matrix/Vector Operations) for the formulas.

The routine D2SPG stores the value of c_j. We get it with a call to the options manager routine SUMAG (Chapter 11, Utilities). A pointer, or offset into the work array, is obtained as an integer option. This gives the location of g and Δg. The solution vector Δy replaces Δg at that location. *Caution*: If a user writes code wherein g is computed with reverse communication and partials are evaluated with divided differences, then there will be *two* distinct places where g is to be stored. This example shows a correct place to get this offset.

This example also serves as a prototype for large, structured (possibly nonlinear) DAE problems where the user must use special methods to store and factor the matrix A and solve the linear system $A\Delta y = \Delta g$. The word “factor” is used literally here. A user could, for instance, solve the system using an iterative method. Generally, the factor step can be any preparatory phase required for a later solve step.

```
      USE IMSL_LIBRARIES
      INTEGER    N
      PARAMETER  (N=10)
!                                 SPECIFICATIONS FOR PARAMETERS
      INTEGER    ICHAP, IGET, INUM, IPUT, IRNUM
      PARAMETER  (ICHAP=5, IGET=1, INUM=6, IPUT=2, IRNUM=7)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
```

```
      INTEGER    I, IDO, IN(50), INR(20), IOPT(6), IVAL(7), IWK(35+N), &
                J, NOUT
      REAL       A(N,N), GVAL(N), H(N,N), SC, SS, SUMY, SVAL(1), T, &
                TEND, WK(41+11*N), Y(N), YPR(N), Z
!                                 SPECIFICATIONS FOR INTRINSICS
      INTRINSIC  ABS, SQRT
      REAL       ABS, SQRT
!                                 SPECIFICATIONS FOR SUBROUTINES
!                                 SPECIFICATIONS FOR FUNCTIONS
      EXTERNAL   DGSPG, DJSPG
!                                 Define initial data
      IDO = 1
      T = 0.0E0
      TEND = 1.0E0
!                                 Initial values
      CALL SSET (N, 1.0E0, Y, 1)
      CALL SSET (N, 0.0, YPR, 1)
!                                 Initial lower Hessenberg matrix
      CALL SSET (N*N, 0.0E0, H, 1)
      DO 20  I=1, N - 1
         DO 10  J=1, I + 1
            H(I,J) = J - I
   10    CONTINUE
   20 CONTINUE
      DO 30  J=1, N
         H(N,J) = J - N
   30 CONTINUE
!                                 Get integer option numbers
      IOPT(1) = INUM
      CALL IUMAG ('math', ICHAP, IGET, 1, IOPT, IN)
!                                 Get floating point option numbers
      IOPT(1) = IRNUM
      CALL IUMAG ('math', ICHAP, IGET, 1, IOPT, INR)
!                                 Set for reverse communication
!                                 evaluation of g.
      IOPT(1) = IN(26)
      IVAL(1) = 0
!                                 Set for evaluation of partial
!                                 derivatives.
      IOPT(2) = IN(5)
      IVAL(2) = 1
!                                 Set for reverse communication
!                                 evaluation of partials.
      IOPT(3) = IN(29)
      IVAL(3) = 0
!                                 Set for reverse communication
!                                 solution of linear equations.
      IOPT(4) = IN(31)
      IVAL(4) = 0
!                                 Storage for the partial
!                                 derivative array not allocated.
      IOPT(5) = IN(34)
      IVAL(5) = 1
!                                 Set the sizes of IWK, WK
!                                 for internal checking.
```

```
      IOPT(6) = IN(35)
      IVAL(6) = 35 + N
      IVAL(7) = 41 + 11*N
!                                 ’Put’ integer options.
      CALL IUMAG (’math’, ICHAP, IPUT, 6, IOPT, IVAL)
!                                 Write problem title.
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99998)
!                                 Integrate ODE/DAE.  Use
!                                 dummy IMSL external names.
   40 CONTINUE
      CALL D2SPG (N, T, TEND, IDO, Y, YPR, DGSPG, DJSPG, IWK, WK)
!                                 Find where g goes.
!                                 (It only goes in one place
!                                 here, but can vary if
!                                 divided differences are used
!                                 for partial derivatives.)
      IOPT(1) = IN(27)
      CALL IUMAG (’math’, ICHAP, IGET, 1, IOPT, IVAL)
!                                 Direct user response.
      GO TO (50, 180, 60, 50, 90, 100, 130, 150), IDO
   50 CONTINUE
!                                 This should not occur.
      WRITE (NOUT,*) ’ Unexpected return with IDO = ’, IDO
   60 CONTINUE
!                                 Reset options to defaults
      DO 70  I=1, 50
         IN(I) = -IN(I)
   70 CONTINUE
      CALL IUMAG (’math’, ICHAP, IPUT, 50, IN, IVAL)
      DO 80  I=1, 20
         INR(I) = -INR(I)
   80 CONTINUE
      CALL UMAG (’math’, ICHAP, IPUT, INR, SVAL, numopts=1)
      STOP
   90 CONTINUE
!                                 Return came for g evaluation.
      CALL SCOPY (N, YPR, 1, GVAL, 1)
      CALL SGEMV (’NO’, N, N, 1.0E0, H, N, Y, 1, -1.0E0, GVAL, 1)
!                                 Put g into place.
      CALL SCOPY (N, GVAL, 1, WK(IVAL(1:)), 1)
      GO TO 40
  100 CONTINUE
!                                 Return came for partial
!                                 derivative evaluation.
  110 CALL SCOPY (N*N, H, 1, A, 1)
!                                 Get value of c_j for partials.
      IOPT(1) = INR(9)
      CALL UMAG (’math’, ICHAP, IGET, IOPT, SVAL, numopts=1)
!                                 Subtract c_j from diagonals
!                                 to compute (partials for y’)*c_j.
      DO 120  I=1, N
         A(I,I) = A(I,I) - SVAL(1)
  120 CONTINUE
      GO TO 40
  130 CONTINUE
!                                 Return came for factorization
      DO 140  J=1, N - 1
!                                 Construct and apply Givens
!                                 transformations.
         CALL SROTG (A(J,J), A(J,J+1), SC, SS)
         CALL SROT (N-J, A((J+1):,1), 1, A((J+1):,J+1), 1, SC, SS)
  140 CONTINUE
      GO TO 40
  150 CONTINUE
!                                 Return came to solve the system
      CALL SCOPY (N, WK(IVAL(1)), 1, GVAL, 1)
      DO 160  J=1, N - 1
         GVAL(J) = GVAL(J)/A(J,J)
         CALL SAXPY (N-J, -GVAL(J), A((J+1):,J), 1, GVAL((J+1):, 1)
  160 CONTINUE
      GVAL(N) = GVAL(N)/A(N,N)
!                                 Reconstruct Givens rotations
      DO 170  J=N - 1, 1, -1
         Z = A(J,J+1)
         IF (ABS(Z) .LT. 1.0E0) THEN
            SC = SQRT(1.0E0-Z**2)
            SS = Z
         ELSE IF (ABS(Z) .GT. 1.0E0) THEN
            SC = 1.0E0/Z
            SS = SQRT(1.0E0-SC**2)
         ELSE
            SC = 0.0E0
            SS = 1.0E0
         END IF
         CALL SROT (1, GVAL(J:), 1, GVAL((J+1):), 1, SC, SS)
  170 CONTINUE
      CALL SCOPY (N, GVAL, 1, WK(IVAL(1)), 1)
      GO TO 40
!
  180 CONTINUE
      SUMY = 0.E0
      DO 190  I=1, N
         SUMY = SUMY + Y(I)
  190 CONTINUE
      WRITE (NOUT,99999) TEND, SUMY
!                                 Finish up internally
      IDO = 3
      GO TO 40
99998 FORMAT (11X, ’T’, 6X, ’Sum of Y(i), i=1,n’)
99999 FORMAT (2F15.5)
      END
```

Output

```
   T      Sum of Y(i), i=1,n
1.00000       65.17058
```

Introduction to Subroutine PDE_1D_MG

The section describes an algorithm and a corresponding integrator subroutine PDE_1D_MG for solving a system of partial differential equations

This software is a one-dimensional solver. It requires initial and boundary conditions in addition to values of u_t. The integration method is noteworthy due to the maintenance of grid lines in the space variable, x. Details for choosing new grid lines are given in Blom and Zegeling, (1994). The class of problems solved with PDE_1D_MG is expressed by equations:

$$\sum_{k=1}^{NPDE} C_{j,k}\left(x,t,u,u_x\right)\frac{\partial u^k}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m R_j\left(x,t,u,u_x\right)\right) - Q_j\left(x,t,u,u_x\right),$$
$$j = 1,\ldots,NPDE, x_L < x < x_{R,} t > t_0, m \in \{0,1,2\}$$

Equation 1

The vector

$$u \equiv \left[u' \ldots, u^{NPDE}\right]^T$$

is the solution. The integer value NPDE ≥ 1 is the number of differential equations. The functions R_j and Qj can be regarded, in special cases, as flux and source terms. The functions

$$u, C_{j,k}, R_j \text{ and } Q_j$$

are expected to be continuous. Allowed values

$$m = 0, m = 1, \text{ and } m = 2$$

are for problems in Cartesian, cylindrical or polar, and spherical coordinates. In the two cases $m > 0$, the interval

$$\left[x_L, x_R\right]$$

must not contain $x = 0$ as an interior point.

The boundary conditions have the master equation form

$$\beta_j\left(x,t\right)R_j\left(x,t,u,u_x\right) = \gamma_j\left(x,t,u,u_x\right),$$
$$\text{at } x = x_L \text{ and } x = x_R, j = 1,\ldots,NPDE$$

Equation 2

In the boundary conditions the

$$\beta_j \text{ and } \gamma_j$$

are continuous functions of their arguments. In the two cases $m > 0$ and an endpoint occurs at 0, the finite value of the solution at $x = 0$ must be ensured. This requires the specification of the solution at $x = 0$, or implies that

$$R_j\Big|_{x=x_L} = 0$$

or

$$R_j\Big|_{x=x_R} = 0 .$$

The initial values satisfy

$$u(x,t_0) = u_0(x), x \in [x_L, x_R],$$

where u_0 is a piece-wise continuous vector function of x with $NPDE$ components.

The user must pose the problem so that mathematical definitions are known for the functions

$$C_{k,j}, R_j, Q_j, \beta_j, \gamma_j \text{ and } u_0.$$

These functions are provided to the routine `PDE_1D_MG` in the form of three subroutines. Optionally, this information can be provided by *reverse communication*. These forms of the interface are explained below and illustrated with examples. Users may turn directly to the examples if they are comfortable with the description of the algorithm.

PDE_1D_MG

Invokes a module, with the statement `USE PDE_1D_MG`, near the second line of the program unit. The integrator is provided with single or double precision arithmetic, and a generic named interface is provided. We do not recommend using 32-bit floating point arithmetic here. The routine is called within the following loop, and is entered with each value of `IDO`. The loop continues until a value of `IDO` results in an exit.

```
IDO=1

DO

   CASE(IDO == 1) {Do required initialization steps}

   CASE(IDO == 2) {Save solution, update T0 and TOUT }

          IF{Finished with integration} IDO=3

   CASE(IDO == 3) EXIT {Normal}

   CASE(IDO == 4) EXIT {Due to errors}

   CASE(IDO == 5) {Evaluate initial data}

   CASE(IDO == 6) {Evaluate differential equations}

   CASE(IDO == 7) {Evaluate boundary conditions}

   CASE(IDO == 8) {Prepare to solve banded system}

   CASE(IDO == 9) {Solve banded system}

   CALL PDE_1D_MG (T0, TOUT, IDO, U,&
   initial_conditions,&
   pde_system_definition,&
   boundary_conditions, IOPT)

END DO
```

The arguments to `PDE_1D_MG` are *required* or *optional.*

Required Arguments

`T0`—(Input/Output)
This is the value of the independent variable t where the integration of u_t begins. It is set to the value `TOUT` on return.

`TOUT`—(Input)
This is the value of the independent variable t where the integration of u_t ends. Note: Values of `T0 < TOUT` imply integration in the forward direction, while values of `T0 > TOUT` imply integration in the backward direction. Either direction is permitted.

`IDO`—(Input/Output)
This in an integer flag that directs program control and user action. Its value is used for initialization, termination, and for directing user response during reverse communication:

`IDO`=1 This value is assigned by the user for the start of a new problem. Internally it causes allocated storage to be reallocated, conforming to the problem size. Various initialization steps are performed.

`IDO`=2 This value is assigned by the routine when the integrator has successfully reached the end point, `TOUT`.

`IDO`=3 This value is assigned by the user at the end of a problem. The routine is called by the user with this value. Internally it causes termination steps to be performed.

`IDO`=4 This value is assigned by the integrator when a type `FATAL` or `TERMINAL` error condition has occurred, and error processing is set **NOT** to **STOP** for these types of errors. It is not necessary to make a final call to the integrator with **`IDO=3`** in this case.

Values of **`IDO =`** **5,6,7,8,9** are reserved for applications that provide problem information or linear algebra computations using reverse communication. When problem information is provided using reverse communication, the differential equations, boundary conditions and initial data must all be given. The absence of optional subroutine names in the calling sequence directs the routine to use reverse communication. In the module `PDE_1D_MG_INT`, scalars and arrays for evaluating results are named below. The names are preceded by the prefix "`s_pde_1d_mg_`" or "`d_pde_1d_mg_`", depending on the precision. We use the prefix "`?_pde_1d_mg_`", for the appropriate choice.

`IDO`=5 This value is assigned by the integrator, requesting data for the initial conditions. Following this evaluation the integrator is re-entered.

(Optional) Update the grid of values in array locations $U(NPDE+1, j)$ j = 2, …, N. This grid is returned to the user equally spaced, but can be updated as desired, provided the values are increasing.

(Required) Provide initial values for all components of the system at the grid of values $U(NPDE+1, j)$ j = 1, …, N. If the optional step of updating the initial grid is performed, then the initial values are evaluated at the updated grid.

IDO=6 This value is assigned by the integrator, requesting data for the differential equations. Following this evaluation the integrator is re-entered. Evaluate the terms of the system of Equation 2. A default value of $m = 0$ is assumed, but this can be changed to one of the other choices $m = 1$ or 2. Use the optional argument IOPT(:) for that purpose. Put the values in the arrays as indicated[1].

$$\begin{aligned}
&x \equiv ?\_pde\_1d\_mg\_x \\
&t \equiv ?\_pde\_1d\_mg\_t \\
&u^j \equiv ?\_pde\_1d\_mg\_u(j) \\
&\frac{\partial u^j}{\partial x} = u_x^j \equiv ?\_pde\_1d\_mg\_dudx(j) \\
&?\_pde\_1d\_mg\_c(j,k) := C_{j,k}(x,t,u,u_x) \\
&?\_pde\_1d\_mg\_r(j) := r_j(x,t,u,u_x) \\
&?\_pde\_1d\_mg\_q(j) := q_j(x,t,u,u_x) \\
&j,k = 1,\ldots,NPDE
\end{aligned}$$

If any of the functions cannot be evaluated, set pde_1d_mg_ires=3. Otherwise do not change its value.

IDO=7 This value is assigned by the integrator, requesting data for the boundary conditions, as expressed in Equation 3. Following the evaluation the integrator is re-entered.

$$\begin{aligned}
&x \equiv ?\_pde\_1d\_mg\_x \\
&t \equiv ?\_pde\_1d\_mg\_t \\
&u^j \equiv ?\_pde\_1d\_mg\_u(j) \\
&\frac{\partial u^j}{\partial x} = u_x^j \equiv ?\_pde\_1d\_mg\_dudx(j) \\
&?\_pde\_1d\_mg\_beta(j) := \beta_j(x,t,u,u_x) \\
&?\_pde\_1d\_mg\_gamma(j) := \gamma_j(x,t,u,u_x) \\
&j = 1,\ldots,NPDE
\end{aligned}$$

[1] The assign-to equality, $a := b$, used here and below, is read “the expression b is evaluated and then assigned to the location a.”

The value $x \in \{x_L, x_R\}$, and the logical flag pde_1d_mg_LEFT=.TRUE. for $x = x_L$. It has the value pde_1d_mg_LEFT=.FALSE. for $x = x_R$. If any of the functions cannot be evaluated, set pde_1d_mg_ires=3. Otherwise do not change its value.

IDO=8 This value is assigned by the integrator, requesting the calling program to prepare for solving a banded linear system of algebraic equations. This value will occur only when the option for "reverse communication solving" is set in the array IOPT(:), with option PDE_1D_MG_REV_COMM_FACTOR_SOLVE. The matrix data for this system is in *Band Storage Mode*, described in the section: Reference Material for the IMSL Fortran Numerical Libraries.

PDE_1D_MG_IBAND	Half band-width of linear system
PDE_1D_MG_LDA	The value 3*PDE_1D_MG_IBAND+1, with $NEQ = (NPDE + 1)N$
?_PDE_1D_MG_A	Array of size PDE_1D_MG_LDA by NEQ holding the problem matrix in *Band Storage Mode*
PDE_1D_MG_PANIC_FLAG	Integer set to a non-zero value only if the linear system is detected as singular

IDO=9 This value is assigned by the integrator , requesting the calling program to solve a linear system with the matrix defined as noted with **IDO=8.**

?_PDE_1D_MG_RHS	Array of size NEQ holding the linear system problem right-hand side
PDE_1D_MG_PANIC_FLAG	Integer set to a non-zero value only if the linear system is singular
?_PDE_1D_MG_SOL	Array of size NEQ to receive the solution, after the solving step

U(1:NPDE+1,1:N)—(Input/Output)

This assumed-shape array specifies *Input* information about the problem size and boundaries. The dimension of the problem is obtained from $NPDE + 1 = size(U,1)$. The number of grid points is obtained by $N = \text{size}(U,2)$. Limits for the variable x are assigned as input in array locations, $U(NPDE+1, 1) = x_L$, $U(NPDE+1, N) = x_R$. It is not required to define $U(NPDE+1, j)$, j=2, …, N-1. At completion, the array U(1:NPDE,1:N) contains the approximate solution value $U_i(x_j(TOUT), TOUT)$ in location U(I,J). The grid value $x_j(TOUT)$ is in location U(NPDE+1,J). Normally the grid values are equally spaced as the integration starts. Variable spaced grid values can be provided by defining them as *Output* from the subroutine initial_conditions or during reverse communication, **IDO=5**.

Optional Arguments

initial_conditions—(Input)
The name of an external subroutine, written by the user, when using forward communication. If this argument is not used, then reverse communication is used to provide the problem information. The routine gives the initial values for the system at the starting independent variable value T0. This routine can also provide a non-uniform grid at the initial value.

```
SUBROUTINE initial_conditions (NPDE,N,U)
  Integer NPDE, N
  REAL(kind(T0)) U(:,:)
END SUBROUTINE
```

(Optional) Update the grid of values in array locations $U(NPDE+1, j), j = 2, \ldots, N-1$. This grid is input equally spaced, but can be updated as desired, provided the values are increasing.

(Required) Provide initial values $U(:, j), j = 1, \ldots, N$ for all components of the system at the grid of values $U(NPDE+1, j), j = 1, \ldots, N$. If the optional step of updating the initial grid is performed, then the initial values are evaluated at the updated grid.

pde_system_definition—(Input)
The name of an external subroutine, written by the user, when using forward communication. It gives the differential equation, as expressed in Equation 2.

```
SUBROUTINE pde_system_definition&
 (t, x, NPDE, u, dudx, c, q, r, IRES)
  Integer NPDE, IRES
  REAL(kind(T0)) t, x, u(:), dudx(:)
  REAL(kind(T0)) c(:,:), q(:), r(:)
END SUBROUTINE
```

Evaluate the terms of the system of . A default value of $m = 0$ is assumed, but this can be changed to one of the other choices $m = 1$ or 2. Use the optional argument IOPT(:) for that purpose. Put the values in the arrays as indicated.

$$
\begin{aligned}
&u^j \equiv u(j) \\
&\frac{\partial u^j}{\partial x} = u_x^j \equiv dudx(j) \\
&c(j,k) := C_{j,k}(x,t,u,u_x) \\
&r(j) := r_j(x,t,u,u_x) \\
&q(j) := q_j(x,t,u,u_x) \\
&j,k = 1,\ldots,NPDE
\end{aligned}
$$

If any of the functions cannot be evaluated, set `IRES=3`. Otherwise do not change its value.

`boundary_conditions`—(Input)
The name of an external subroutine, written by the user when using forward communication. It gives the boundary conditions, as expressed in Equation 2.

$$
\begin{aligned}
&u^j \equiv u(j) \\
&\frac{\partial u^j}{\partial x} = u_x^j \equiv dudx(j) \\
&beta(j) := \beta_j(x,t,u,u_x) \\
&gamma(j) := \gamma_j(x,t,u,u_x) \\
&j = 1,\ldots,NPDE
\end{aligned}
$$

The value $x \in \{x_L, x_R\}$, and the logical flag `LEFT=.TRUE.` for $x = x_L$. The flag has the value `LEFT=.FALSE.` for $x = x_R$.

`IOPT`—(Input)
Derived type array `s_options` or `d_options`, used for passing optional data to `PDE_1D_MG`. See the section **Optional Data** in the **Introduction** for an explanation of the derived type and its use. It is necessary to invoke a module, with the statement `USE ERROR_OPTION_PACKET`, near the second line of the program unit. Examples 2-8 use this optional argument. The choices are as follows:

Packaged Options for PDE_1D_MG

Option Prefix = ?	Option Name	Option Value
`S_`, `d_`	`PDE_1D_MG_CART_COORDINATES`	1
`S_`, `d_`	`PDE_1D_MG_CYL_COORDINATES`	2
`S_`, `d_`	`PDE_1D_MG_SPH_COORDINATES`	3
`S_`, `d_`	`PDE_1D_MG_TIME_SMOOTHING`	4
`S_`, `d_`	`PDE_1D_MG_SPATIAL_SMOOTHING`	5
`S_`, `d_`	`PDE_1D_MG_MONITOR_REGULARIZING`	6
`S_`, `d_`	`PDE_1D_MG_RELATIVE_TOLERANCE`	7
`S_`, `d_`	`PDE_1D_MG_ABSOLUTE_TOLERANCE`	8

Packaged Options for PDE_1D_MG		
s_, d_	PDE_1D_MG_MAX_BDF_ORDER	9
s_, d_	PDE_1D_MG_REV_COMM_FACTOR_SOLVE	10
s_, d_	PDE_1D_MG_NO_NULLIFY_STACK	11

IOPT(IO) = PDE_1D_MG_CART_COORDINATES
Use the value $m = 0$ in Equation 2. This is the default.

IOPT(IO) = PDE_1D_MG_CYL_COORDINATES
Use the value $m = 1$ in Equation 2. The default value is $m = 0$.

IOPT(IO) = PDE_1D_MG_SPH_COORDINATES
Use the value $m = 2$ in Equation 2. The default value is $m = 0$.

IOPT(IO) =
?_OPTIONS(PDE_1D_MG_TIME_SMOOTHING,TAU)
This option resets the value of the parameter $\tau \geq 0$, described above.
The default value is $\tau = 0$.

IOPT(IO) =
?_OPTIONS(PDE_1D_MG_SPATIAL_SMOOTHING,KAP)
This option resets the value of the parameter $\kappa \geq 0$, described above.
The default value is $\kappa = 2$.

IOPT(IO) =
?_OPTIONS(PDE_1D_MG_MONITOR_REGULARIZING,ALPH)
This option resets the value of the parameter $\alpha \geq 0$, described above.
The default value is $\alpha = 0.01$.

IOPT(IO) = ?_OPTIONS
(PDE_1D_MG_RELATIVE_TOLERANCE,RTOL)
This option resets the value of the relative accuracy parameter used in DASPG. The default value is RTOL=1E-2 for single precision and RTOL=1D-4 for double precision.

IOPT(IO) = ?_OPTIONS
(PDE_1D_MG_ABSOLUTE_TOLERANCE,ATOL)
This option resets the value of the absolute accuracy parameter used in DASPG. The default value is ATOL=1E-2 for single precision and ATOL=1D-4 for double precision.

IOPT(IO) = PDE_1D_MG_MAX_BDF_ORDER
IOPT(IO+1) = MAXBDF
Reset the maximum order for the BDF formulas used in DASPG. The default value is MAXBDF=2. The new value can be any integer between 1 and 5. Some problems will benefit by making this change. We used the default value due to the fact that DASPG may cycle on its selection of order and step-size with orders higher than value 2.

IOPT(IO) = PDE_1D_MG_REV_COMM_FACTOR_SOLVE

The calling program unit will solve the banded linear systems required in the stiff differential-algebraic equation integrator. Values of **IDO=8, 9** will occur only when this optional value is used.

IOPT(IO) = PDE_1D_MG_NO_NULLIFY_STACK

To maintain an efficient interface, the routine PDE_1D_MG collapses the subroutine call stack with CALL_E1PSH("NULLIFY_STACK"). This implies that the overhead of maintaining the stack will be eliminated, which may be important with reverse communication. It does not eliminate error processing. However, precise information of which routines have errors will not be displayed. To see the full call chain, this option should be used. Following completion of the integration, stacking is turned back on with CALL_E1POP("NULLIFY_STACK").

FORTRAN 90 Interface

Generic: CALL PDE_1D_MG (T0, TOUT, IDO, [,…])

Specific: The specific interface names are S_PDE_1D_MG and D_PDE_1D_MG.

Remarks on the Examples

Due to its importance and the complexity of its interface, this subroutine is presented with several examples. Many of the program features are exercised. The problems complete without any change to the optional arguments, except where these changes are required to describe or to solve the problem.

In many applications the solution to a PDE is used as an auxiliary variable, perhaps as part of a larger design or simulation process. The truncation error of the approximate solution is commensurate with piece-wise linear interpolation on the grid of values, at each output point. To show that the solution is reasonable, a graphical display is revealing and helpful. We have not provided graphical output as part of our documentation, but users may already have the Visual Numerics, Inc. product, PV-WAVE, not included with Fortran 90 MP Library. Examples 1-8 write results in files "PDE_ex0?.out" that can be visualized with PV-WAVE. We provide a script of commands, "pde_1d_mg_plot.pro", for viewing the solutions (see example below). The grid of values and each consecutive solution component is displayed in separate plotting windows. The script and data files written by examples 1-8 on a SUN-SPARC system are in the directory for Fortran 90 MP Library examples. When inside PV_WAVE, execute the command line "pde_1d_mg_plot,filename='PDE_ex0?.out'" to view the output of a particular example.

Code for PV-WAVE Plotting (Examples Directory)

```
PRO PDE_1d_mg_plot, FILENAME = filename, PAUSE = pause
;
   if keyword_set(FILENAME) then file = filename else file = "res.dat"
   if keyword_set(PAUSE) then twait = pause else twait = .1
;
;       Define floating point variables that will be read
;       from the first line of the data file.
```

```
   xl = 0D0
   xr = 0D0
   t0 = 0D0
   tlast = 0D0
;
;       Open the data file and read in the problem parameters.
   openr, lun, filename, /get_lun
   readf, lun, npde, np, nt, xl, xr, t0, tlast

;       Define the arrays for the solutions and grid.
   u = dblarr(nt, npde, np)
   g = dblarr(nt, np)
   times = dblarr(nt)
;
;       Define a temporary array for reading in the data.
   tmp = dblarr(np)
   t_tmp = 0D0
;
;       Read in the data.
   for i = 0, nt-1 do begin     ; For each step in time
    readf, lun, t_tmp
    times(i) = t_tmp

    for k = 0, npde-1 do begin  ;    For each PDE:
       rmf, lun, tmp
       u(i,k,*) = tmp           ;    Read in the components.
    end

    rmf, lun, tmp
    g(i,*) = tmp                ;    Read in the grid.
   end
;
;       Close the data file and free the unit.
   close, lun
   free_lun, lun
;
;       We now have all of the solutions and grids.
;
;       Delete any window that is currently open.
   while (!d.window NE -1) do WDELETE
;
;       Open two windows for plotting the solutions
;       and grid.
   window, 0, xsize = 550, ysize = 420
   window, 1, xsize = 550, ysize = 420
;
;        Plot the grid.
   wset, 0
   plot, [xl, xr], [t0, tlast], /nodata, ystyle = 1, $
         title = "Grid Points", xtitle = "X", ytitle = "Time"
   for i = 0, np-1 do begin
      oplot, g(*, i), times, psym = -1
   end
;
;       Plot the solution(s):
   wset, 1
   for k = 0, npde-1 do begin
      umin = min(u(*,k,*))
      umax = max(u(*,k,*))
      for i = 0, nt-1 do begin
         title = strcompress("U_"+string(k+1), /remove_all)+ $
                 " at time "+string(times(i))
         plot, g(i, *), u(i,k,*), ystyle = 1, $
               title = title, xtitle = "X", $
               ytitle = strcompress("U_"+string(k+1), /remove_all), $
               xr = [xl, xr], yr = [umin, umax], $
               psym = -4
         wait, twait
      end
   end

end
```

Example 1 - Electrodynamics Model

This example is from Blom and Zegeling (1994). The system is

$$\begin{aligned}
&u_t \varepsilon p u_{xx} - g(u-v)\\
&v_t p v_{xx} + g(u-v),\\
&\text{where } g(z) = exp(\eta z/3) - exp(-2\eta z/3)\\
&0 \le x \le 1, 0 \le t \le 4\\
&u = 1 \text{ and } v = 0 \text{ at } t = 0\\
&u_x = 0 \text{ and } v = 0 \text{ at } x = 0\\
&u = 1 \text{ and } v_x = 0 \text{ at } x = 1\\
&\varepsilon = 0.143, p = 0.1743, \eta = 17.19
\end{aligned}$$

We make the connection between the model problem statement and the example:

$$\begin{aligned}
&C = I_2\\
&m = 0, R_1 = \varepsilon p u_x, R_2 = p v_x\\
&Q_1 = g(u-v), Q_2 = -Q_1
\end{aligned}$$

The boundary conditions are

$$\begin{aligned}
&\beta_1 = 1, \beta_2 = 0, \gamma_1 = 0, \gamma_2 = v, \text{ at } x = x_L = 0\\
&\beta_1 = 0, \beta_2 = 1, \gamma_1 = u - 1, \gamma_2 = 0, \text{ at } x = x_R = 1
\end{aligned}$$

Rationale: Example 1

This is a non-linear problem with sharply changing conditions near $t = 0$. The default settings of integration parameters allow the problem to be solved. The use of `PDE_1D_MG` with forward communication requires three subroutines provided by the user to describe the initial conditions, differential equations, and boundary conditions.

```
     program PDE_EX1
! Electrodynamics Model:
        USE PDE_1d_mg
        IMPLICIT NONE
        INTEGER, PARAMETER :: NPDE=2, N=51, NFRAMES=5
        INTEGER I, IDO
! Define array space for the solution.
        real(kind(1d0)) U(NPDE+1,N), T0, TOUT
        real(kind(1d0)) :: ZERO=0D0, ONE=1D0, &
          DELTA_T=10D0, TEND=4D0
        EXTERNAL IC_01, PDE_01, BC_01
! Start loop to integrate and write solution values.
        IDO=1
        DO
           SELECT CASE (IDO)
! Define values that determine limits.
           CASE (1)
              T0=ZERO
              TOUT=1D-3
              U(NPDE+1,1)=ZERO;U(NPDE+1,N)=ONE
              OPEN(FILE='PDE_ex01.out',UNIT=7)
              WRITE(7, "(3I5, 4F10.5)") NPDE, N, NFRAMES,&
                U(NPDE+1,1), U(NPDE+1,N), T0, TEND
! Update to the next output point.
! Write solution and check for final point.
           CASE (2)
              WRITE(7,"(F10.5)")TOUT
              DO I=1,NPDE+1
                WRITE(7,"(4E15.5)")U(I,:)
              END DO
              T0=TOUT;TOUT=TOUT*DELTA_T
              IF(T0 >= TEND) IDO=3
              TOUT=MIN(TOUT, TEND)
! All completed.  Solver is shut down.
           CASE (3)
              CLOSE(UNIT=7)
              EXIT
           END SELECT
! Forward communication is used for the problem data.
           CALL PDE_1D_MG (T0, TOUT, IDO, U,&
             initial_conditions= IC_01,&
             PDE_system_definition= PDE_01,&
             boundary_conditions= BC_01)
        END DO
     END
     SUBROUTINE IC_01(NPDE, NPTS, U)
! This is the initial data for Example 1.
        IMPLICIT NONE
        INTEGER NPDE, NPTS
        REAL(KIND(1D0)) U(NPDE+1,NPTS)
        U(1,:)=1D0;U(2,:)=0D0
     END SUBROUTINE
     SUBROUTINE PDE_01(T, X, NPDE, U, DUDX, C, Q, R, IRES)
! This is the differential equation for Example 1.
        IMPLICIT NONE
        INTEGER NPDE, IRES
        REAL(KIND(1D0)) T, X, U(NPDE), DUDX(NPDE),&
          C(NPDE,NPDE), Q(NPDE), R(NPDE)
        REAL(KIND(1D0)) :: EPS=0.143D0, P=0.1743D0,&
          ETA=17.19D0, Z, TWO=2D0, THREE=3D0
        C=0D0;C(1,1)=1D0;C(2,2)=1D0
        R=P*DUDX;R(1)=R(1)*EPS
        Z=ETA*(U(1)-U(2))/THREE
        Q(1)=EXP(Z)-EXP(-TWO*Z)
        Q(2)=-Q(1)
     END SUBROUTINE
     SUBROUTINE BC_01(T, BTA, GAMA, U, DUDX, NPDE, LEFT, IRES)
! These are the boundary conditions for Example 1.
        IMPLICIT NONE
        INTEGER NPDE, IRES
        LOGICAL LEFT
        REAL(KIND(1D0)) T, BTA(NPDE), GAMA(NPDE),&
          U(NPDE), DUDX(NPDE)
        IF(LEFT) THEN
           BTA(1)=1D0;BTA(2)=0D0
           GAMA(1)=0D0;GAMA(2)=U(2)
        ELSE
           BTA(1)=0D0;BTA(2)=1D0
           GAMA(1)=U(1)-1D0;GAMA(2)=0D0
        END IF
     END SUBROUTINE
```

Description

The equation

$$u_t = f(u,x,t), x_L < x < x_R, t > t_0,$$

is approximated at N time-dependent grid values

$$x_L = x_0 < \ldots < x_i\left(t\right) < x_{i+1}\left(t\right) < \ldots < x_N = x_R.$$

Using the total differential

$$\frac{du}{dt} = u_t + u_x \frac{dx}{dt}$$

transforms the differential equation to

$$\frac{du}{dt} - u_x \frac{dx}{dt} = u_t = f\left(u,x,t\right).$$

Using central divided differences for the factor u_x leads to the system of ordinary differential equations in implicit form

$$\frac{dU_i}{dt} - \frac{\left(U_{i+1} - U_{i-1}\right)}{\left(x_{i+1} - x_{i-1}\right)} \frac{dx_i}{dt} = F_i, t > t_0, i = 1, \ldots, N$$
.

The terms U_i, F_i respectively represent the approximate solution to the partial differential equation and the value of $f(u,x,t)$ at the point $(x,t) = (x_i,(t),t)$. The truncation error is second-order in the space variable, x. The above ordinary differential equations are underdetermined, so additional equations are added for the variation of the time-dependent grid points. It is necessary to discuss these equations, since they contain parameters that can be adjusted by the user. Often it will be necessary to modify these parameters to solve a difficult problem. For this purpose the following quantities are defined[2]:

$$\begin{aligned}
&\Delta x_i = x_{i+1} - x_i, n_i = \left(\Delta x_i\right)^{-1} \\
&\mu_i = n_i - \kappa\left(\kappa + 1\right)\left(n_{i+1} - 2n_i + n_{i-1}\right), 0 \le i \le N \\
&n_{-1} \equiv n_0, n_{N+1} \equiv n_N
\end{aligned}$$

The values n_i are the so-called point concentration of the grid, and $\kappa \ge 0$ denotes a spatial smoothing parameter. Now the grid points are defined implicitly so that

$$\frac{\mu_{i-1} + \tau \frac{d\mu_{i-1}}{dt}}{M_{i-1}} = \frac{\mu_i + \tau \frac{d\mu_1}{dt}}{M_i}, 1 \le i \le N,$$

where $\tau \ge 1$ is a time-smoothing parameter. Choosing τ very large results in a fixed grid. Increasing the value of τ from its default avoids the error condition where grid lines cross. The divisors are

$$M_i^2 = \alpha + \left(NPDE\right)^{-1} \sum_{j=1}^{NPDE} \frac{\left(U_{i+1}^j - U_i^j\right)^2}{\left(\Delta x_i\right)^2}$$
.

The value κ determines the level of clustering or spatial smoothing of the grid points. Decreasing κ from its default decrease the amount of spatial smoothing. The parameters M_i approximate arc length and help determine the shape of the grid or x_i-distribution. The parameter τprevents the grid movement from adjusting immediately to new values of the M_i, thereby avoiding oscillations in the grid that cause large relative errors. This is important when applied to solutions with steep gradients.

The discrete form of the differential equation and the smoothing equations are combined to yield the implicit system of differential equations

[2] The three-tiered equal sign, used here and below, is read "$a \equiv b$ or a and b are exactly the same object or value."

$$A(Y)\frac{dY}{dt} = L(Y),$$
$$Y = \left[U_1^1,\ldots,U_1^{NPDE},x_1,\ldots,U_j^1,\ldots,U_j^{NPDE},x_j,\ldots\right]^T$$

This is frequently a stiff differential-algebraic system. It is solved using the integrator `DASPG` and its subroutines, including `D2SPG`. These are documented in this chapter. Note that `DASPG` is restricted to use within `PDE_1D_MG` until the routine exits with the flag `IDO = 3`. If `DASPG` is needed during the evaluations of the differential equations or boundary conditions, use of a second processor and inter-process communication is required. The only options for `DASPG` set by `PDE_1D_MG` are the Maximum BDF Order, and the absolute and relative error values, `ATOL` and `RTOL`. Users may set other options using the Options Manager. This is described in routine `DASPG`, see page 889, and generally in Chapter 11 of this manual.

Additional Examples

Example 2 - Inviscid Flow on a Plate

This example is a first order system from Pennington and Berzins, (1994). The equations are

$$\begin{aligned}
&u_t = -v_x\\
&uu_t = -vu_x + w_x\\
&w = u_x, \text{ implying that } uu_t = -vu_x + u_{xx}\\
&u(0,t) = v(0,t) = 0, u(\infty,t) \equiv u(x_R,t) = 1, t \ge 0\\
&u(x,0) = 1, v(x,0) = 0, x \ge 0
\end{aligned}$$

Following elimination of w, there remain $NPDE = 2$ differential equations. The variable t is not time, but a second space variable. The integration goes from $t = 0$ to $t = 5$. It is necessary to truncate the variable x at a finite value, say $x_{max} = x_R = 25$. In terms of the integrator, the system is defined by letting $m = 0$ and

$$C = \{C_{jk}\} = \begin{bmatrix} 1 & 0 \\ u & 0 \end{bmatrix}, R = \begin{bmatrix} -v \\ u_x \end{bmatrix}, Q = \begin{bmatrix} 0 \\ vu_x \end{bmatrix}$$

The boundary conditions are satisfied by

$$\beta = 0, \gamma = \begin{bmatrix} u - exp(-20t) \\ v \end{bmatrix}, \text{ at } x = x_L$$
$$\beta = 0, \gamma = \begin{bmatrix} u - 1 \\ v_x \end{bmatrix}, \text{at } x = x_R$$

We use $N = 10 + 51 = 61$ grid points and output the solution at steps of $\Delta t = 0.1$.

Rationale: Example 2

This is a non-linear boundary layer problem with sharply changing conditions near $t = 0$. The problem statement was modified so that boundary conditions are continuous near $t = 0$. Without this change the underlying integration software, DASPG, cannot solve the problem. The continuous blending function $u - exp(-20t)$ is arbitrary and artfully chosen. This is a mathematical change to the problem, required because of the stated discontinuity at $t = 0$. Reverse communication is used for the problem data. No additional user-written subroutines are required when using reverse communication. We also have chosen 10 of the initial grid points to be concentrated near $x_L = 0$, anticipating rapid change in the solution near that point. Optional changes are made to use a pure absolute error tolerance and non-zero time-smoothing.

```
        program PDE_1D_MG_EX02
! Inviscid Flow Over a Plate
        USE PDE_1d_mg
        USE ERROR_OPTION_PACKET
        IMPLICIT NONE

        INTEGER, PARAMETER :: NPDE=2, N1=10, N2=51, N=N1+N2
        INTEGER I, IDO, NFRAMES
! Define array space for the solution.
        real(kind(1d0)) U(NPDE+1,N), T0, TOUT, DX1, DX2, DIF
        real(kind(1d0)) :: ZERO=0D0, ONE=1D0, DELTA_T=1D-1,&
          TEND=5D0, XMAX=25D0
        real(kind(1d0)) :: U0=1D0, U1=0D0, TDELTA=1D-1, TOL=1D-2
        TYPE(D_OPTIONS) IOPT(3)
! Start loop to integrate and record solution values.
        IDO=1
        DO
           SELECT CASE (IDO)
! Define values that determine limits and options.
           CASE (1)
              T0=ZERO
              TOUT=DELTA_T
              U(NPDE+1,1)=ZERO;U(NPDE+1,N)=XMAX
              OPEN(FILE='PDE_ex02.out',UNIT=7)
              NFRAMES=NINT((TEND+DELTA_T)/DELTA_T)
              WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES,&
                U(NPDE+1,1), U(NPDE+1,N), T0, TEND
              DX1=XMAX/N2;DX2=DX1/N1
              IOPT(1)=D_OPTIONS(PDE_1D_MG_RELATIVE_TOLERANCE,ZERO)
              IOPT(2)=D_OPTIONS(PDE_1D_MG_ABSOLUTE_TOLERANCE,TOL)
              IOPT(3)=D_OPTIONS(PDE_1D_MG_TIME_SMOOTHING,1D-3)

! Update to the next output point.
! Write solution and check for final point.
           CASE (2)
              T0=TOUT
              IF(T0 <= TEND) THEN
                 WRITE(7,"(F10.5)")TOUT
                 DO I=1,NPDE+1
                    WRITE(7,"(4E15.5)")U(I,:)
                 END DO
                 TOUT=MIN(TOUT+DELTA_T,TEND)
                 IF(T0 == TEND)IDO=3
              END IF
```

```
! All completed.  Solver is shut down.
           CASE (3)

              CLOSE(UNIT=7)
              EXIT

! Define initial data values.
           CASE (5)
              U(:NPDE,:)=ZERO;U(1,:)=ONE
              DO I=1,N1
                 U(NPDE+1,I)=(I-1)*DX2
              END DO
              DO I=N1+1,N
                 U(NPDE+1,I)=(I-N1)*DX1
              END DO
              WRITE(7,"(F10.5)")T0
              DO I=1,NPDE+1
                 WRITE(7,"(4E15.5)")U(I,:)
              END DO

! Define differential equations.
           CASE (6)
              D_PDE_1D_MG_C=ZERO
              D_PDE_1D_MG_C(1,1)=ONE
              D_PDE_1D_MG_C(2,1)=D_PDE_1D_MG_U(1)

              D_PDE_1D_MG_R(1)=-D_PDE_1D_MG_U(2)
              D_PDE_1D_MG_R(2)= D_PDE_1D_MG_DUDX(1)

              D_PDE_1D_MG_Q(1)= ZERO
              D_PDE_1D_MG_Q(2)= &
                D_PDE_1D_MG_U(2)*D_PDE_1D_MG_DUDX(1)
! Define boundary conditions.
           CASE (7)
              D_PDE_1D_MG_BETA=ZERO
              IF(PDE_1D_MG_LEFT) THEN
                 DIF=EXP(-20D0*D_PDE_1D_MG_T)
! Blend the left boundary value down to zero.
                 D_PDE_1D_MG_GAMMA=(/D_PDE_1D_MG_U(1)-DIF,D_PDE_1D_MG_U(2)/)
              ELSE
                 D_PDE_1D_MG_GAMMA=(/D_PDE_1D_MG_U(1)-
ONE,D_PDE_1D_MG_DUDX(2)/)
              END IF
           END SELECT

! Reverse communication is used for the problem data.
           CALL PDE_1D_MG (T0, TOUT, IDO, U, IOPT=IOPT)
        END DO
     end program
```

Example 3 - Population Dynamics

This example is from Pennington and Berzins (1994). The system is

$$u_t = -u_x - I(t)u,\ x_L = 0 \le x \le a = x_R,\ t \ge 0$$

$$I(t) = \int_0^a u(x,t)dx$$

$$u(x,0) = \frac{exp(-x)}{2 - exp(-a)}$$

$$u(0,t) = g\left(\int_0^a b(x,I(t))u(x,t)dx, t\right), \text{ where}$$

$$b(x,y) = \frac{xy\,exp(-x)}{(y+1)^2}, \text{ and}$$

$$g(z,t) =$$

$$\frac{4z(2 - 2\,exp(-a) + exp(-t))^2}{(1 - exp(-a))(1 - (1+2a)exp(-2a))(1 - exp(-a) + exp(-t))}$$

This is a notable problem because it involves the unknown $u(x,t) = \frac{exp(-x)}{1 - exp(-a) + exp(-t)}$ across the entire domain. The software can solve the problem by introducing two dependent algebraic equations:

$$v_1(t) = \int_0^a u(x,t)dx,$$

$$v_2(t) = \int_0^a x\,exp(-x)u(x,t)dx$$

This leads to the modified system

$$u_t = -u_x - v_1 u,\ 0 \le x \le a,\ t \ge 0$$

$$u(0,t) = \frac{g(1,t)v_1 v_2}{(v_1+1)^2}$$

In the interface to the evaluation of the differential equation and boundary conditions, it is necessary to evaluate the integrals, which are computed with the values of $u(x,t)$ on the grid. The integrals are approximated using the trapezoid rule, commensurate with the truncation error in the integrator.

Rationale: Example 3

This is a non-linear integro-differential problem involving non-local conditions for the differential equation and boundary conditions. Access to evaluation of these conditions is provided using reverse communication. It is not possible to solve this problem with forward communication, given the current subroutine interface. Optional changes are made to use an absolute error tolerance and non-zero time-smoothing. The time-smoothing value $\tau = 1$ prevents grid lines from crossing.

```
     program PDE_1D_MG_EX03
! Population Dynamics Model.
        USE PDE_1d_mg
        USE ERROR_OPTION_PACKET
        IMPLICIT NONE
        INTEGER, PARAMETER :: NPDE=1, N=101
        INTEGER IDO, I, NFRAMES
! Define array space for the solution.
        real(kind(1d0)) U(NPDE+1,N), MID(N-1), T0, TOUT, V_1, V_2
        real(kind(1d0)) :: ZERO=0D0, HALF=5D-1, ONE=1D0,&
          TWO=2D0, FOUR=4D0, DELTA_T=1D-1,TEND=5D0, A=5D0
        TYPE(D_OPTIONS) IOPT(3)
! Start loop to integrate and record solution values.
        IDO=1
        DO
           SELECT CASE (IDO)
! Define values that determine limits.
           CASE (1)
              T0=ZERO
              TOUT=DELTA_T
              U(NPDE+1,1)=ZERO;U(NPDE+1,N)=A
              OPEN(FILE='PDE_ex03.out',UNIT=7)
              NFRAMES=NINT((TEND+DELTA_T)/DELTA_T)
              WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES,&
                U(NPDE+1,1), U(NPDE+1,N), T0, TEND
              IOPT(1)=D_OPTIONS(PDE_1D_MG_RELATIVE_TOLERANCE,ZERO)
              IOPT(2)=D_OPTIONS(PDE_1D_MG_ABSOLUTE_TOLERANCE,1D-2)
              IOPT(3)=D_OPTIONS(PDE_1D_MG_TIME_SMOOTHING,1D0)
! Update to the next output point.
! Write solution and check for final point.
           CASE (2)
              T0=TOUT
              IF(T0 <= TEND) THEN
                WRITE(7,"(F10.5)")TOUT
                DO I=1,NPDE+1
                  WRITE(7,"(4E15.5)")U(I,:)
                END DO
                TOUT=MIN(TOUT+DELTA_T,TEND)
                IF(T0 == TEND)IDO=3
              END IF
! All completed.  Solver is shut down.
           CASE (3)
              CLOSE(UNIT=7)
              EXIT
! Define initial data values.
           CASE (5)
              U(1,:)=EXP(-U(2,:))/(TWO-EXP(-A))
              WRITE(7,"(F10.5)")T0
              DO I=1,NPDE+1
                WRITE(7,"(4E15.5)")U(I,:)
              END DO
! Define differential equations.
```

```
           CASE (6)
              D_PDE_1D_MG_C(1,1)=ONE
              D_PDE_1D_MG_R(1)=-D_PDE_1D_MG_U(1)
! Evaluate the approximate integral, for this t.
              V_1=HALF*SUM((U(1,1:N-1)+U(1,2:N))*&
                          (U(2,2:N) - U(2,1:N-1)))
              D_PDE_1D_MG_Q(1)=V_1*D_PDE_1D_MG_U(1)
! Define boundary conditions.
           CASE (7)
              IF(PDE_1D_MG_LEFT) THEN
! Evaluate the approximate integral, for this t.
! A second integral is needed at the edge.
              V_1=HALF*SUM((U(1,1:N-1)+U(1,2:N))*&
                          (U(2,2:N) - U(2,1:N-1)))
              MID=HALF*(U(2,2:N)+U(2,1:N-1))
              V_2=HALF*SUM(MID*EXP(-MID)*&
              (U(1,1:N-1)+U(1,2:N))*(U(2,2:N)-U(2,1:N-1)))
                 D_PDE_1D_MG_BETA=ZERO

D_PDE_1D_MG_GAMMA=G(ONE,D_PDE_1D_MG_T)*V_1*V_2/(V_1+ONE)**2-&
                   D_PDE_1D_MG_U
              ELSE
                 D_PDE_1D_MG_BETA=ZERO
                 D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_DUDX(1)
              END IF
            END SELECT
! Reverse communication is used for the problem data.
           CALL PDE_1D_MG (T0, TOUT, IDO, U, IOPT=IOPT)
        END DO
CONTAINS
        FUNCTION G(z,t)
        IMPLICIT NONE
          REAL(KIND(1d0)) Z, T, G
          G=FOUR*Z*(TWO-TWO*EXP(-A)+EXP(-T))**2
          G=G/((ONE-EXP(-A))*(ONE-(ONE+TWO*A)*&
            EXP(-TWO*A))*(1-EXP(-A)+EXP(-T)))
        END FUNCTION
     end program
```

Example 4 - A Model in Cylindrical Coordinates

This example is from Blom and Zegeling (1994). The system models a reactor-diffusion problem:

$$T_z = r^{-1}\frac{\partial(\beta r T_r)}{\partial r} + \gamma\, exp\left(\frac{T}{1+\varepsilon T}\right)$$

$$T_r(0,z) = 0,\, T(1,z) = 0,\, z > 0$$

$$T(r,0) = 0,\, 0 \le r < 1$$

$$\beta = 10^{-4}, \gamma = 1, \varepsilon = 0.1$$

The axial direction z is treated as a time coordinate. The radius r is treated as the single space variable.

Rationale: Example 4

This is a non-linear problem in cylindrical coordinates. Our example illustrates assigning $m = 1$ in Equation 2. We provide an optional argument that resets this value from its default, $m = 0$. Reverse communication is used to interface with the problem data.

```
      program PDE_1D_MG_EX04
! Reactor-Diffusion problem in cylindrical coordinates.
        USE pde_1d_mg
        USE error_option_packet
        IMPLICIT NONE
        INTEGER, PARAMETER :: NPDE=1, N=41
        INTEGER IDO, I, NFRAMES
! Define array space for the solution.
        real(kind(1d0)) T(NPDE+1,N), Z0, ZOUT
        real(kind(1d0)) :: ZERO=0D0, ONE=1D0, DELTA_Z=1D-1,&
          ZEND=1D0, ZMAX=1D0, BTA=1D-4, GAMA=1D0, EPS=1D-1
        TYPE(D_OPTIONS) IOPT(1)
! Start loop to integrate and record solution values.
        IDO=1
        DO
           SELECT CASE (IDO)
! Define values that determine limits.
           CASE (1)
              Z0=ZERO
              ZOUT=DELTA_Z
              T(NPDE+1,1)=ZERO;T(NPDE+1,N)=ZMAX
              OPEN(FILE='PDE_ex04.out',UNIT=7)
              NFRAMES=NINT((ZEND+DELTA_Z)/DELTA_Z)
              WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES,&
                T(NPDE+1,1), T(NPDE+1,N), Z0, ZEND
              IOPT(1)=PDE_1D_MG_CYL_COORDINATES
! Update to the next output point.
! Write solution and check for final point.
           CASE (2)
              IF(Z0 <= ZEND) THEN
                WRITE(7,"(F10.5)")ZOUT
                DO I=1,NPDE+1
                  WRITE(7,"(4E15.5)")T(I,:)
                END DO
                ZOUT=MIN(ZOUT+DELTA_Z,ZEND)
                IF(Z0 == ZEND)IDO=3
              END IF
! All completed.  Solver is shut down.
           CASE (3)
              CLOSE(UNIT=7)
              EXIT
! Define initial data values.
           CASE (5)
              T(1,:)=ZERO
              WRITE(7,"(F10.5)")Z0
              DO I=1,NPDE+1
                WRITE(7,"(4E15.5)")T(I,:)
              END DO
! Define differential equations.
```

```
           CASE (6)
              D_PDE_1D_MG_C(1,1)=ONE
              D_PDE_1D_MG_R(1)=BTA*D_PDE_1D_MG_DUDX(1)
              D_PDE_1D_MG_Q(1)= -GAMA*EXP(D_PDE_1D_MG_U(1)/&
                (ONE+EPS*D_PDE_1D_MG_U(1)))
! Define boundary conditions.
           CASE (7)
              IF(PDE_1D_MG_LEFT) THEN
                 D_PDE_1D_MG_BETA=ONE; D_PDE_1D_MG_GAMMA=ZERO
              ELSE
                 D_PDE_1D_MG_BETA=ZERO; D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_U(1)
              END IF
           END SELECT
! Reverse communication is used for the problem data.
! The optional derived type changes the internal model
! to use cylindrical coordinates.
           CALL PDE_1D_MG (Z0, ZOUT, IDO, T, IOPT=IOPT)
        END DO
     end program
```

Example 5 - A Flame Propagation Model

This example is presented more fully in Verwer, *et al.*, (1989). The system is a normalized problem relating mass density $u(x,t)$ and temperature $v(x,t)$:

$$
\begin{aligned}
& u_t = u_{xx} - uf(v) \\
& v_t = v_{xx} + uf(v), \\
& \text{where } f(z) = \gamma\, exp(-\beta / z), \beta = 4, \gamma = 3.52 \times 10^6 \\
& 0 \le x \le 1, 0 \le t \le 0.006 \\
& u(x,0) = 1, v(x,0) = 0.2 \\
& u_x = v_x = 0, x = 0 \\
& u_x = 0, v = b(t), x = 1, \text{ where} \\
& b(t) = 1.2, \text{ for } t \ge 2 \times 10^{-4}, \text{ and} \\
& =\ 0.2 + 5 \times 10^3 t, \text{ for } 0 \le t \le 2 \times 10^{-4}
\end{aligned}
$$

Rationale: Example 5

This is a non-linear problem. The example shows the model steps for replacing the banded solver in the software with one of the user's choice. Reverse communication is used for the interface to the problem data and the linear solver. Following the computation of the matrix factorization in DL2CRB, we declare the system to be singular when the reciprocal of the condition number is smaller than the working precision. This choice is not suitable for all problems. Attention must be given to detecting a singularity when this option is used.

```
     program PDE_1D_MG_EX05
! Flame propagation model
        USE pde_1d_mg
        USE ERROR_OPTION_PACKET
        USE Numerical_Libraries, ONLY :&
         dl2crb, dlfsrb
        IMPLICIT NONE

        INTEGER, PARAMETER :: NPDE=2, N=40, NEQ=(NPDE+1)*N
        INTEGER I, IDO, NFRAMES, IPVT(NEQ)

! Define array space for the solution.
        real(kind(1d0)) U(NPDE+1,N), T0, TOUT
! Define work space for the banded solver.
        real(kind(1d0)) WORK(NEQ), RCOND
        real(kind(1d0)) :: ZERO=0D0, ONE=1D0, DELTA_T=1D-4,&
          TEND=6D-3, XMAX=1D0, BTA=4D0, GAMA=3.52D6
        TYPE(D_OPTIONS) IOPT(1)
! Start loop to integrate and record solution values.
        IDO=1
        DO
           SELECT CASE (IDO)

! Define values that determine limits.
           CASE (1)
              T0=ZERO
              TOUT=DELTA_T
              U(NPDE+1,1)=ZERO; U(NPDE+1,N)=XMAX
              OPEN(FILE='PDE_ex05.out',UNIT=7)
              NFRAMES=NINT((TEND+DELTA_T)/DELTA_T)
              WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES,&
                U(NPDE+1,1), U(NPDE+1,N), T0, TEND
              IOPT(1)=PDE_1D_MG_REV_COMM_FACTOR_SOLVE
! Update to the next output point.
! Write solution and check for final point.
           CASE (2)
             T0=TOUT
              IF(T0 <= TEND) THEN
                WRITE(7,"(F10.5)")TOUT
                DO I=1,NPDE+1
                  WRITE(7,"(4E15.5)")U(I,:)
                END DO
                TOUT=MIN(TOUT+DELTA_T,TEND)
                IF(T0 == TEND)IDO=3
              END IF

! All completed.  Solver is shut down.
           CASE (3)
              CLOSE(UNIT=7)
              EXIT

! Define initial data values.
           CASE (5)
              U(1,:)=ONE; U(2,:)=2D-1
              WRITE(7,"(F10.5)")T0
              DO I=1,NPDE+1
                WRITE(7,"(4E15.5)")U(I,:)
              END DO
! Define differential equations.
```

```
           CASE (6)
              D_PDE_1D_MG_C=ZERO
              D_PDE_1D_MG_C(1,1)=ONE; D_PDE_1D_MG_C(2,2)=ONE

              D_PDE_1D_MG_R=D_PDE_1D_MG_DUDX

              D_PDE_1D_MG_Q(1)=  D_PDE_1D_MG_U(1)*F(D_PDE_1D_MG_U(2))
              D_PDE_1D_MG_Q(2)= -D_PDE_1D_MG_Q(1)
! Define boundary conditions.
           CASE (7)
              IF(PDE_1D_MG_LEFT) THEN
                 D_PDE_1D_MG_BETA=ZERO;D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_DUDX
              ELSE
                 D_PDE_1D_MG_BETA(1)=ONE
                 D_PDE_1D_MG_GAMMA(1)=ZERO
                 D_PDE_1D_MG_BETA(2)=ZERO
                 IF(D_PDE_1D_MG_T >= 2D-4) THEN
                   D_PDE_1D_MG_GAMMA(2)=12D-1
                 ELSE
                   D_PDE_1D_MG_GAMMA(2)=2D-1+5D3*D_PDE_1D_MG_T
                 END IF
                 D_PDE_1D_MG_GAMMA(2)=D_PDE_1D_MG_GAMMA(2)-&
                  D_PDE_1D_MG_U(2)
              END IF
           CASE(8)
! Factor the banded matrix.  This is the same solver used
! internally but that is not required.  A user can substitute
! one of their own.
             call dl2crb (neq, d_pde_1d_mg_a, pde_1d_mg_lda,
pde_1d_mg_iband,&
               pde_1d_mg_iband, d_pde_1d_mg_a, pde_1d_mg_lda, ipvt, rcond,
work)
             IF(rcond <= EPSILON(ONE)) pde_1d_mg_panic_flag = 1
           CASE(9)
! Solve using the factored banded matrix.
             call dlfsrb(neq, d_pde_1d_mg_a, pde_1d_mg_lda,
pde_1d_mg_iband,&
               pde_1d_mg_iband, ipvt, d_pde_1d_mg_rhs, 1, d_pde_1d_mg_sol)
           END SELECT

! Reverse communication is used for the problem data.
           CALL PDE_1D_MG (T0, TOUT, IDO, U, IOPT=IOPT)
        END DO
CONTAINS
        FUNCTION F(Z)
        IMPLICIT NONE
        REAL(KIND(1D0)) Z, F
          F=GAMA*EXP(-BTA/Z)
        END FUNCTION
     end program
```

Example 6 - A ‘Hot Spot’ Model

This example is presented more fully in Verwer, *et al.*, (1989). The system is a normalized problem relating the temperature $u(x,t)$, of a reactant in a chemical system. The formula for $h(z)$ is equivalent to their example.

$$
\begin{aligned}
&u_t = u_{xx} + h(u),\\
&\text{where } h(z) = \frac{R}{a\delta}(1 + a - z)\,exp(-\delta(1/z-1)),\\
&a = 1, \delta = 20, R = 5\\
&0 \le x \le 1, 0 \le t \le 0.29\\
&u(x,0) = 1\\
&u_x = 0, x = 0\\
&u = 1, x = 1
\end{aligned}
$$

Rationale: Example 6

This is a non-linear problem. The output shows a case where a rapidly changing front, or hot-spot, develops after a considerable way into the integration. This causes rapid change to the grid. An option sets the maximum order BDF formula from its default value of 2 to the theoretical stable maximum value of 5.

```
      USE pde_1d_mg
      USE error_option_packet
      IMPLICIT NONE

      INTEGER, PARAMETER :: NPDE=1, N=80
      INTEGER I, IDO, NFRAMES

! Define array space for the solution.
      real(kind(1d0)) U(NPDE+1,N), T0, TOUT
      real(kind(1d0)) :: ZERO=0D0, ONE=1D0, DELTA_T=1D-2,&
        TEND=29D-2, XMAX=1D0, A=1D0, DELTA=2D1, R=5D0
      TYPE(D_OPTIONS) IOPT(2)
! Start loop to integrate and record solution values.
      IDO=1
      DO
         SELECT CASE (IDO)

! Define values that determine limits.
         CASE (1)
            T0=ZERO
            TOUT=DELTA_T
            U(NPDE+1,1)=ZERO; U(NPDE+1,N)=XMAX
            OPEN(FILE='PDE_ex06.out',UNIT=7)
            NFRAMES=(TEND+DELTA_T)/DELTA_T
            WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES,&
              U(NPDE+1,1), U(NPDE+1,N), T0, TEND
! Illustrate allowing the BDF order to increase
! to its maximum allowed value.
```

```
              IOPT(1)=PDE_1D_MG_MAX_BDF_ORDER
                IOPT(2)=5
! Update to the next output point.
! Write solution and check for final point.
           CASE (2)
              T0=TOUT
              IF(T0 <= TEND) THEN
                WRITE(7,"(F10.5)")TOUT
                DO I=1,NPDE+1
                  WRITE(7,"(4E15.5)")U(I,:)
                END DO
                TOUT=MIN(TOUT+DELTA_T,TEND)
                IF(T0 == TEND)IDO=3
              END IF
! All completed.  Solver is shut down.
           CASE (3)
              CLOSE(UNIT=7)
              EXIT

! Define initial data values.
           CASE (5)
              U(1,:)=ONE
              WRITE(7,"(F10.5)")T0
              DO I=1,NPDE+1
                WRITE(7,"(4E15.5)")U(I,:)
              END DO
! Define differential equations.
           CASE (6)
              D_PDE_1D_MG_C=ONE
              D_PDE_1D_MG_R=D_PDE_1D_MG_DUDX
              D_PDE_1D_MG_Q= - H(D_PDE_1D_MG_U(1))

! Define boundary conditions.
           CASE (7)
              IF(PDE_1D_MG_LEFT) THEN
                 D_PDE_1D_MG_BETA=ZERO
                 D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_DUDX
              ELSE

                 D_PDE_1D_MG_BETA=ZERO
                 D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_U(1)-ONE
              END IF
           END SELECT

! Reverse communication is used for the problem data.
           CALL PDE_1D_MG (T0, TOUT, IDO, U, IOPT=IOPT)
        END DO
CONTAINS
        FUNCTION H(Z)
        real(kind(1d0)) Z, H
          H=(R/(A*DELTA))*(ONE+A-Z)*EXP(-DELTA*(ONE/Z-ONE))
        END FUNCTION
     end program
```

Example 7 - Traveling Waves

This example is presented more fully in Verwer, *et al.*, (1989). The system is a normalized problem relating the interaction of two waves, $u(x,t)$ and $v(x,t)$ moving in opposite directions. The waves meet and reduce in amplitude, due to the non-linear terms in the equation. Then they separate and travel onward, with reduced amplitude.

$$
\begin{aligned}
&u_t = -u_x - 100uv,\\
&v_t = v_x - 100uv,\\
&-0.5 \le x \le 0.5,\, 0 \le t \le 0.5\\
&u(x,0) = 0.5\bigl(1 + cos(10\pi x)\bigr), x \in [-0.3, -0.1], \text{ and}\\
&\qquad = 0, \text{ otherwise}\\
&v(x,0) = 0.5\bigl(1 + cos(10\pi x)\bigr), x \in [0.1, 0.3], \text{ and}\\
&\qquad = 0, \text{ otherwise}\\
&u = v = 0 \text{ at both ends, } t \ge 0
\end{aligned}
$$

Rationale: Example 7

This is a non-linear system of first order equations.

```
     program PDE_1D_MG_EX07
! Traveling Waves
        USE pde_1d_mg
        USE error_option_packet
        IMPLICIT NONE

        INTEGER, PARAMETER :: NPDE=2, N=50
        INTEGER I, IDO, NFRAMES

! Define array space for the solution.
        real(kind(1d0)) U(NPDE+1,N), TEMP(N), T0, TOUT
        real(kind(1d0)) :: ZERO=0D0, HALF=5D-1, &
          ONE=1D0, DELTA_T=5D-2,TEND=5D-1, PI
        TYPE(D_OPTIONS) IOPT(5)
! Start loop to integrate and record solution values.
        IDO=1
        DO
           SELECT CASE (IDO)

! Define values that determine limits.
           CASE (1)
              T0=ZERO
              TOUT=DELTA_T
              U(NPDE+1,1)=-HALF; U(NPDE+1,N)=HALF
              OPEN(FILE='PDE_ex07.out',UNIT=7)
              NFRAMES=(TEND+DELTA_T)/DELTA_T
              WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES,&
                U(NPDE+1,1), U(NPDE+1,N), T0, TEND
              IOPT(1)=D_OPTIONS(PDE_1D_MG_TIME_SMOOTHING,1D-3)
              IOPT(2)=D_OPTIONS(PDE_1D_MG_RELATIVE_TOLERANCE,ZERO)
              IOPT(3)=D_OPTIONS(PDE_1D_MG_ABSOLUTE_TOLERANCE,1D-3)
              IOPT(4)=PDE_1D_MG_MAX_BDF_ORDER
                IOPT(5)=3
! Update to the next output point.
! Write solution and check for final point.
           CASE (2)
              T0=TOUT
              IF(T0 <= TEND) THEN
                WRITE(7,"(F10.5)")TOUT
                DO I=1,NPDE+1
                  WRITE(7,"(4E15.5)")U(I,:)
                END DO
                TOUT=MIN(TOUT+DELTA_T,TEND)
                IF(T0 == TEND)IDO=3
              END IF

! All completed.  Solver is shut down.
           CASE (3)
              CLOSE(UNIT=7)
              EXIT

! Define initial data values.
           CASE (5)
              TEMP=U(3,:)
              U(1,:)=PULSE(TEMP); U(2,:)=U(1,:)
              WHERE (TEMP < -3D-1 .or. TEMP > -1D-1) U(1,:)=ZERO
              WHERE (TEMP <  1D-1 .or. TEMP >  3D-1) U(2,:)=ZERO
              WRITE(7,"(F10.5)")T0
              DO I=1,NPDE+1
                WRITE(7,"(4E15.5)")U(I,:)
              END DO

! Define differential equations.
           CASE (6)
              D_PDE_1D_MG_C=ZERO
              D_PDE_1D_MG_C(1,1)=ONE; D_PDE_1D_MG_C(2,2)=ONE

              D_PDE_1D_MG_R=D_PDE_1D_MG_U
              D_PDE_1D_MG_R(1)=-D_PDE_1D_MG_R(1)

              D_PDE_1D_MG_Q(1)= 100D0*D_PDE_1D_MG_U(1)*D_PDE_1D_MG_U(2)
              D_PDE_1D_MG_Q(2)= D_PDE_1D_MG_Q(1)

! Define boundary conditions.
           CASE (7)
              D_PDE_1D_MG_BETA=ZERO;D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_U

           END SELECT

! Reverse communication is used for the problem data.
           CALL PDE_1D_MG (T0, TOUT, IDO, U, IOPT=IOPT)
        END DO
CONTAINS
        FUNCTION PULSE(Z)
        real(kind(1d0)) Z(:), PULSE(SIZE(Z))
```

```
          PI=ACOS(-ONE)
          PULSE=HALF*(ONE+COS(10D0*PI*Z))
        END FUNCTION
     end program
```

Example 8 - Black-Scholes

The value of a European “call option,” $c(s,t)$, with exercise price e and expiration date T, satisfies the “asset-or-nothing payoff” $c(s,T) = s, s \geq e; = 0, s < e$. Prior to expiration $c(s,t)$ is estimated by the Black-Scholes differential equation $c_t + \frac{\sigma^2}{2}s^2c_{ss} + rsc_s - rc \equiv c_t + \frac{\sigma^2}{2}\left(s^2c_s\right)_s + \left(r-\sigma^2\right)sc_s - rc = 0$. The parameters in the model are the risk-free interest rate, r, and the stock volatility, σ. The boundary conditions are $c(0,t) = 0$ and $c_s(s,t) \approx 1, s \to \infty$. This development is described in Wilmott, *et al.* (1995), pages 41-57. There are explicit solutions for this equation based on the Normal Curve of Probability. The normal curve, and the solution itself, can be efficiently computed with the IMSL function ANORDF, IMSL (1994), page 186. With numerical integration the equation itself or the payoff can be readily changed to include other formulas, $c(s,T)$, and corresponding boundary conditions. We use $e = 100, r = 0.08, T - t = 0.25, \sigma^2 = 0.04, s_L = 0,$ and $s_R = 150$.

Rationale: Example 8

This is a linear problem but with initial conditions that are discontinuous. It is necessary to use a positive time-smoothing value to prevent grid lines from crossing. We have used an absolute tolerance of 10^{-3}. In $US, this is one-tenth of a cent.

```
     program PDE_1D_MG_EX08
! Black-Scholes call price
        USE pde_1d_mg
        USE error_option_packet
        IMPLICIT NONE

        INTEGER, PARAMETER :: NPDE=1, N=100
        INTEGER I, IDO, NFRAMES

! Define array space for the solution.
        real(kind(1d0)) U(NPDE+1,N), T0, TOUT, SIGSQ, XVAL
        real(kind(1d0)) :: ZERO=0D0, HALF=5D-1, ONE=1D0, &
          DELTA_T=25D-3, TEND=25D-2, XMAX=150, SIGMA=2D-1, &
          R=8D-2, E=100D0
        TYPE(D_OPTIONS) IOPT(5)
! Start loop to integrate and record solution values.
        IDO=1
        DO
           SELECT CASE (IDO)

! Define values that determine limits.
           CASE (1)
              T0=ZERO
              TOUT=DELTA_T
              U(NPDE+1,1)=ZERO; U(NPDE+1,N)=XMAX
              OPEN(FILE='PDE_ex08.out',UNIT=7)
              NFRAMES=NINT((TEND+DELTA_T)/DELTA_T)
              WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES,&
                U(NPDE+1,1), U(NPDE+1,N), T0, TEND
                SIGSQ=SIGMA**2
! Illustrate allowing the BDF order to increase
! to its maximum allowed value.
              IOPT(1)=PDE_1D_MG_MAX_BDF_ORDER
                IOPT(2)=5
              IOPT(3)=D_OPTIONS(PDE_1D_MG_TIME_SMOOTHING,5D-3)
              IOPT(4)=D_OPTIONS(PDE_1D_MG_RELATIVE_TOLERANCE,ZERO)
              IOPT(5)=D_OPTIONS(PDE_1D_MG_ABSOLUTE_TOLERANCE,1D-2)
! Update to the next output point.
! Write solution and check for final point.
           CASE (2)
              T0=TOUT
              IF(T0 <= TEND) THEN
                WRITE(7,"(F10.5)")TOUT
                DO I=1,NPDE+1
                  WRITE(7,"(4E15.5)")U(I,:)
                END DO
                TOUT=MIN(TOUT+DELTA_T,TEND)
                IF(T0 == TEND)IDO=3
              END IF
! All completed.  Solver is shut down.
           CASE (3)
              CLOSE(UNIT=7)
              EXIT

! Define initial data values.
           CASE (5)
              U(1,:)=MAX(U(NPDE+1,:)-E,ZERO)  ! Vanilla European Call
              U(1,:)=U(NPDE+1,:)              ! Asset-or-nothing Call
              WHERE(U(1,:) <= E) U(1,:)=ZERO  ! on these two lines
              WRITE(7,"(F10.5)")T0
              DO I=1,NPDE+1
                WRITE(7,"(4E15.5)")U(I,:)
              END DO
! Define differential equations.
           CASE (6)
              XVAL=D_PDE_1D_MG_X
              D_PDE_1D_MG_C=ONE
              D_PDE_1D_MG_R=D_PDE_1D_MG_DUDX*XVAL**2*SIGSQ*HALF
              D_PDE_1D_MG_Q=-(R-SIGSQ)*XVAL*D_PDE_1D_MG_DUDX+R*D_PDE_1D_MG_U
! Define boundary conditions.
           CASE (7)
              IF(PDE_1D_MG_LEFT) THEN
                 D_PDE_1D_MG_BETA=ZERO
                 D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_U
              ELSE

                 D_PDE_1D_MG_BETA=ZERO
                 D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_DUDX(1)-ONE
              END IF
           END SELECT

! Reverse communication is used for the problem data.
           CALL PDE_1D_MG (T0, TOUT, IDO, U, IOPT=IOPT)
        END DO

     end program
```

Example 9 - Electrodynamics, Parameters Studied with MPI

This example, described above in Example 1, is from Blom and Zegeling (1994). The system parameters $\varepsilon, p,$ and η, are varied, using uniform random numbers. The intervals studied are $0.1 \le \varepsilon \le 0.2, 0.1 \le p \le 0.2,$ and $10 \le \eta \le 20$. Using $N = 21$ grid values and other program options, the elapsed time, parameter values, and the value $v(x,t)\big|_{x=1,t=4}$ are sent to the root node. This information is written on a file. The final summary includes the minimum value of

$$v(x,t)\big|_{x=1,t=4},$$

and the maximum and average time per integration, per node.

Rationale: Example 9

This is a non-linear simulation problem. Using at least two integrating processors and MPI allows more values of the parameters to be studied in a given time than with a single processor. This code is valuable as a study guide when an application needs to estimate timing and other output parameters. The simulation time is controlled at the root node. An integration is started, after receiving results, within the first SIM_TIME seconds. The elapsed time will be longer than SIM_TIME by the slowest processor's time for its last integration.

```
     program PDE_1D_MG_EX09
! Electrodynamics Model, parameter study.
        USE PDE_1d_mg
        USE MPI_SETUP_INT
        USE RAND_INT
        USE SHOW_INT
        IMPLICIT NONE
        INCLUDE "mpif.h"
        INTEGER, PARAMETER :: NPDE=2, N=21
        INTEGER I, IDO, IERROR, CONTINUE, STATUS(MPI_STATUS_SIZE)
        INTEGER, ALLOCATABLE :: COUNTS(:)
! Define array space for the solution.
        real(kind(1d0)) :: U(NPDE+1,N), T0, TOUT
        real(kind(1d0)) :: ZERO=0D0, ONE=1D0,DELTA_T=10D0, TEND=4D0
! SIM_TIME is the number of seconds to run the simulation.
        real(kind(1d0)) :: EPS, P, ETA, Z, TWO=2D0, THREE=3D0,
SIM_TIME=60D0
        real(kind(1d0)) :: TIMES, TIMEE, TIMEL, TIME, TIME_SIM,
V_MIN, DATA(5)
        real(kind(1d0)), ALLOCATABLE :: AV_TIME(:), MAX_TIME(:)
        TYPE(D_OPTIONS) IOPT(4), SHOW_IOPT(2)
        TYPE(S_OPTIONS) SHOW_INTOPT(2)
        MP_NPROCS=MP_SETUP(1)
        MPI_NODE_PRIORITY=(/(I-1,I=1,MP_NPROCS)/)
! If NP_NPROCS=1, the program stops.  Change
! MPI_ROOT_WORKS=.TRUE. if MP_NPROCS=1.
        MPI_ROOT_WORKS=.FALSE.
        IF(.NOT. MPI_ROOT_WORKS .and. MP_NPROCS == 1) STOP
        ALLOCATE(AV_TIME(MP_NPROCS), MAX_TIME(MP_NPROCS),
COUNTS(MP_NPROCS))
! Get time start for simulation timing.
        TIME=MPI_WTIME()
        IF(MP_RANK == 0) OPEN(FILE='PDE_ex09.out',UNIT=7)
 SIMULATE: DO
! Pick random parameter values.
           EPS=1D-1*(ONE+rand(EPS))
           P=1D-1*(ONE+rand(P))
           ETA=10D0*(ONE+rand(ETA))
! Start loop to integrate and communicate solution times.
           IDO=1
! Get time start for each new problem.
           DO
              IF(.NOT. MPI_ROOT_WORKS .and. MP_RANK == 0) EXIT
              SELECT CASE (IDO)
! Define values that determine limits.
              CASE (1)
                 T0=ZERO
                 TOUT=1D-3
                 U(NPDE+1,1)=ZERO;U(NPDE+1,N)=ONE
                 IOPT(1)=PDE_1D_MG_MAX_BDF_ORDER
                 IOPT(2)=5
                 IOPT(3)=D_OPTIONS(PDE_1D_MG_RELATIVE_TOLERANCE,1D-2)
                 IOPT(4)=D_OPTIONS(PDE_1D_MG_ABSOLUTE_TOLERANCE,1D-2)

                 TIMES=MPI_WTIME()
! Update to the next output point.
! Write solution and check for final point.
              CASE (2)
                 T0=TOUT;TOUT=TOUT*DELTA_T
                 IF(T0 >= TEND) IDO=3
                 TOUT=MIN(TOUT, TEND)
! All completed.  Solver is shut down.
              CASE (3)
                 TIMEE=MPI_WTIME()
                 EXIT
! Define initial data values.
              CASE (5)
                 U(1,:)=1D0;U(2,:)=0D0
! Define differential equations.
              CASE (6)

D_PDE_1D_MG_C=0D0;D_PDE_1D_MG_C(1,1)=1D0;D_PDE_1D_MG_C(2,2)=1D0
                 D_PDE_1D_MG_R=P*D_PDE_1D_MG_DUDX
D_PDE_1D_MG_R(1)=D_PDE_1D_MG_R(1)*EPS
                 Z=ETA*(D_PDE_1D_MG_U(1)-D_PDE_1D_MG_U(2))/THREE
                 D_PDE_1D_MG_Q(1)=EXP(Z)-EXP(-TWO*Z)
                 D_PDE_1D_MG_Q(2)=-D_PDE_1D_MG_Q(1)
! Define boundary conditions.
              CASE (7)
                 IF(PDE_1D_MG_LEFT) THEN
                    D_PDE_1D_MG_BETA(1)=1D0;D_PDE_1D_MG_BETA(2)=0D0

D_PDE_1D_MG_GAMMA(1)=0D0;D_PDE_1D_MG_GAMMA(2)=D_PDE_1D_MG_U(2)
                 ELSE
                    D_PDE_1D_MG_BETA(1)=0D0;D_PDE_1D_MG_BETA(2)=1D0
                    D_PDE_1D_MG_GAMMA(1)=D_PDE_1D_MG_U(1)-
1D0;D_PDE_1D_MG_GAMMA(2)=0D0
                 END IF
              END SELECT
! Reverse communication is used for the problem data.
              CALL PDE_1D_MG (T0, TOUT, IDO, U)
           END DO
           TIMEL=TIMEE-TIMES
           DATA=(/EPS, P, ETA, U(2,N), TIMEL/)
           IF(MP_RANK > 0) THEN
! Send parameters and time to the root.
              CALL MPI_SEND(DATA, 5, MPI_DOUBLE_PRECISION,0, MP_RANK,
MP_LIBRARY_WORLD, IERROR)
! Receive back a "go/stop" flag.
              CALL MPI_RECV(CONTINUE, 1, MPI_INTEGER, 0, MPI_ANY_TAG,
MP_LIBRARY_WORLD, STATUS, IERROR)
! If root notes that time is up, it sends node a quit flag.
              IF(CONTINUE == 0) EXIT SIMULATE
           ELSE
! If root is working, record its result and then stand ready
! for other nodes to send.
              IF(MPI_ROOT_WORKS) WRITE(7,*) MP_RANK, DATA
! If all nodes have reported, then quit.
              IF(COUNT(MPI_NODE_PRIORITY >= 0) == 0) EXIT SIMULATE
! See if time is up. Some nodes still must report.
              IF(MPI_WTIME()-TIME >= SIM_TIME) THEN
                 CONTINUE=0
              ELSE
                 CONTINUE=1
              END IF
! Root receives simulation data and finds which node sent it.
              IF(MP_NPROCS > 1) THEN
                 CALL MPI_RECV(DATA, 5,
MPI_DOUBLE_PRECISION,MPI_ANY_SOURCE, MPI_ANY_TAG, MP_LIBRARY_WORLD,
STATUS, IERROR)
                 WRITE(7,*) STATUS(MPI_SOURCE), DATA
! If time at the root has elapsed, nodes receive signal to stop.
! Send the reporting node the "go/stop" flag.
! Mark if a node has been stopped.
                 CALL MPI_SEND(CONTINUE, 1, MPI_INTEGER,
STATUS(MPI_SOURCE), 0, MP_LIBRARY_WORLD, IERROR)
                 IF (CONTINUE == 0)
MPI_NODE_PRIORITY(STATUS(MPI_SOURCE)+1) =-
MPI_NODE_PRIORITY(STATUS(MPI_SOURCE)+1)-1
              END IF
              IF (CONTINUE == 0) MPI_NODE_PRIORITY(1)=-1
           END IF
        END DO SIMULATE
        IF(MP_RANK == 0) THEN
           ENDFILE(UNIT=7);REWIND(UNIT=7)
! Read the data. Find extremes and averages.
           MAX_TIME=ZERO;AV_TIME=ZERO;COUNTS=0;V_MIN=HUGE(ONE)
           DO
              READ(7,*, END=10) I, DATA
              COUNTS(I+1)=COUNTS(I+1)+1
              AV_TIME(I+1)=AV_TIME(I+1)+DATA(5)
              IF(MAX_TIME(I+1) < DATA(5)) MAX_TIME(I+1)=DATA(5)
              V_MIN=MIN(V_MIN, DATA(4))
           END DO
10         CONTINUE
           CLOSE(UNIT=7)
! Set printing Index to match node numbering.
           SHOW_IOPT(1)= SHOW_STARTING_INDEX_IS
           SHOW_IOPT(2)=0
           SHOW_INTOPT(1)=SHOW_STARTING_INDEX_IS
           SHOW_INTOPT(2)=0
           CALL SHOW(MAX_TIME,"Maximum Integration Time, per
process:",IOPT=SHOW_IOPT)
           AV_TIME=AV_TIME/MAX(1,COUNTS)
           CALL SHOW(AV_TIME,"Average Integration Time, per
process:",IOPT=SHOW_IOPT)
           CALL SHOW(COUNTS,"Number of
Integrations",IOPT=SHOW_INTOPT)
           WRITE(*,"(1x,A,F6.3)") "Minimum value for v(x,t),at
x=1,t=4:  ",V_MIN
        END IF
        MP_NPROCS=MP_SETUP("Final")
     end program
```

MOLCH

Solves a system of partial differential equations of the form $u_t = f(x, t, u, u_x, u_{xx})$ using the method of lines. The solution is represented with cubic Hermite polynomials.

Required Arguments

IDO — Flag indicating the state of the computation. (Input/Output)

IDO	**State**
1	Initial entry
2	Normal reentry
3	Final call, release workspace

Normally, the initial call is made with IDO = 1. The routine then sets IDO = 2, and this value is then used for all but the last call that is made with IDO = 3.

FCNUT — User-supplied SUBROUTINE to evaluate the function u_t. The usage is CALL FCNUT (NPDES, X, T, U, UX, UXX, UT), where
NPDES – Number of equations. (Input)
X – Space variable, x. (Input)
T – Time variable, t. (Input)
U – Array of length NPDES containing the dependent variable values, u. (Input)
UX – Array of length NPDES containing the first derivatives $u_{x.}$ (Input)
UXX – Array of length NPDES containing the second derivative u_{xx}. (Input)
UT – Array of length NPDES containing the computed derivatives, u_t. (Output)

The name FCNUT must be declared EXTERNAL in the calling program.

FCNBC — User-supplied SUBROUTINE to evaluate the boundary conditions. The boundary conditions accepted by MOLCH are $\alpha_k u_k + \beta_k u_x \equiv \gamma_k$. Note: Users must supply the values α_k and β_k, which determine the values γ_k. Since the γ_k can depend on t, values of γ'_k are also required. Users must supply these values. The usage is CALL FCNBC (NPDES, X, T, ALPHA, BTA, GAMMAP), where

NPDES – Number of equations. (Input)
X – Space variable, x. This value directs which boundary condition to compute. (Input)
T – Time variable, t. (Input)

ALPHA – Array of length NPDES containing the α_k values. (Output)

BTA – Array of length NPDES containing the β_k values. (Output)

GAMMAP – Array of length NPDES containing the values of the derivatives, $\frac{d\gamma_k}{dt} = \gamma_k'$ (Output)

The name FCNBC must be declared EXTERNAL in the calling program.

T — Independent variable, *t*. (Input/Output)
On input, T supplies the initial time, t_0. On output, T is set to the value to which the integration has been updated. Normally, this new value is TEND.

TEND — Value of $t = tend$ at which the solution is desired. (Input)

XBREAK — Array of length NX containing the break points for the cubic Hermite splines used in the *x* discretization. (Input)
The points in the array XBREAK must be strictly increasing. The values XBREAK(1) and XBREAK(NX) are the endpoints of the interval.

Y — Array of size NPDES by NX containing the solution. (Input/Output)
The array Y contains the solution as Y(*k*, *i*) = $u_k(x, tend)$ at *x* = XBREAK(*i*). On input, Y contains the initial values. It ***MUST*** satisfy the boundary conditions. On output, Y contains the computed solution.
There is an optional application of MOLCH that uses derivative values, $u_x(x, t_0)$. The user allocates twice the space for Y to pass this information. The optional derivative information is input as

$$\mathrm{Y}\left(\mathrm{k}, \mathrm{i} + \mathrm{NX}\right) = \frac{\partial u_k}{\partial x}\left(x, t_0\right)$$

at *x* = X(*i*). The array Y contains the optional derivative values as output:

$$\mathrm{Y}\left(\mathrm{k}, \mathrm{i} + \mathrm{NX}\right) = \frac{\partial u_k}{\partial x}\left(x, tend\right)$$

at *x* = X(*i*). To signal that this information is provided, use an options manager call as outlined in Comment 3 and illustrated in Examples 3 and 4.

Optional Arguments

NPDES — Number of differential equations. (Input)
Default: NPDES = size (Y,1).

NX — Number of mesh points or lines. (Input)
Default: NX = size (Y,2).

TOL — Differential equation error tolerance. (Input)
An attempt is made to control the local error in such a way that the global relative error is proportional to TOL.
Default: TOL = 100. * machine precision.

HINIT — Initial step size in the t integration. (Input)
This value must be nonnegative. If HINIT is zero, an initial step size of $0.001|tend - t_0|$ will be arbitrarily used. The step will be applied in the direction of integration.
Default: HINIT = 0.0.

LDY — Leading dimension of Y exactly as specified in the dimension statement of the calling program. (Input)
Default: LDY = size (Y,1).

FORTRAN 90 Interface

Generic: `CALL MOLCH (IDO, FCNUT, FCNBC, T, TEND, XBREAK, Y [,…])`

Specific: The specific interface names are S_MOLCH and D_MOLCH.

FORTRAN 77 Interface

Single:
```
CALL MOLCH (IDO, FCNUT, FCNBC, NPDES, T, TEND, NX, XBREAK,
            TOL, HINIT, Y, LDY)
```

Double: The double precision name is DMOLCH.

Example 1

The normalized linear diffusion PDE, $u_t = u_{xx}$, $0 \le x \le 1$, $t > t_0$, is solved. The initial values are $t_0 = 0$, $u(x, t_0) = u_0 = 1$. There is a “zero-flux” boundary condition at $x = 1$, namely $u_x(1, t) = 0$, $(t > t_0)$. The boundary value of $u(0, t)$ is abruptly changed from u_0 to the value $u_1 = 0.1$. This transition is completed by $t = t_\delta = 0.09$.

Due to restrictions in the type of boundary conditions sucessfully processed by MOLCH, it is necessary to provide the derivative boundary value function γ' at $x = 0$ and at $x = 1$. The function γ at $x = 0$ makes a smooth transition from the value u_0 at $t = t_0$ to the value u_1 at $t = t_\delta$. We compute the transition phase for γ' by evaluating a cubic interpolating polynomial. For this purpose, the function subprogram CSDER, see Chapter 3, Interpolation and Approximation, is used. The interpolation is performed as a first step in the user-supplied routine FCNBC. The function and derivative values $\gamma(t_0) = u_0$, $\gamma'(t_0) = 0$, $\gamma(t_\delta) = u_1$, and $\gamma'(t_\delta) = 0$, are used as input to routine C2HER, to obtain the coefficients evaluated by CSDER. Notice that $\gamma'(t) = 0$, $t > t_\delta$. The evaluation routine CSDER will not yield this value so logic in the routine FCNBC assigns $\gamma'(t) = 0$, $t > t_\delta$.

```
USE MOLCH_INT
USE UMACH_INT
```

```
      USE AMACH_INT
      USE WRRRN_INT
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    LDY, NPDES, NX
      PARAMETER  (NPDES=1, NX=8, LDY=NPDES)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    I, IDO, J, NOUT, NSTEP
      REAL       HINIT, PREC, T, TEND, TOL, XBREAK(NX), Y(LDY,NX)
      CHARACTER  TITLE*19
!                                 SPECIFICATIONS FOR INTRINSICS
      INTRINSIC  FLOAT
      REAL       FLOAT
!                                 SPECIFICATIONS FOR SUBROUTINES
!                                 SPECIFICATIONS FOR FUNCTIONS
      EXTERNAL   FCNBC, FCNUT
!                                 Set breakpoints and initial
!                                 conditions
      U0 = 1.0
      DO 10  I=1, NX
         XBREAK(I) = FLOAT(I-1)/(NX-1)
         Y(1,I)    = U0
   10 CONTINUE
!                                 Set parameters for MOLCH
      PREC = AMACH(4)
      TOL   = SQRT(PREC)
      HINIT = 0.01*TOL
      T     = 0.0
      IDO   = 1
      NSTEP = 10
      CALL UMACH (2, NOUT)
      J = 0
   20 CONTINUE
      J    = J + 1
      TEND = FLOAT(J)/FLOAT(NSTEP)
!                                 This puts more output for small
!                                 t values where action is fastest.
      TEND = TEND**2
!                                 Solve the problem
      CALL MOLCH (IDO, FCNUT, FCNBC, T, TEND, XBREAK, Y, TOL=TOL, HINIT=HINIT)
      IF (J .LE. NSTEP) THEN
!                                 Print results
         WRITE (TITLE,’(A,F4.2)’) ’Solution at T =’, T
         CALL WRRRN (TITLE, Y)
!                                 Final call to release workspace
         IF (J .EQ. NSTEP) IDO = 3
         GO TO 20
      END IF
      END
      SUBROUTINE FCNUT (NPDES, X, T, U, UX, UXX, UT)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NPDES
      REAL       X, T, U(*), UX(*), UXX(*), UT(*)
!
!                                 Define the PDE
      UT(1) = UXX(1)
      RETURN
      END

      SUBROUTINE FCNBC (NPDES, X, T, ALPHA, BTA, GAMP)
      USE CSDER_INT
      USE C2HER_INT
      USE WRRRN_INT
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NPDES
      REAL       X, T, ALPHA(*), BTA(*), GAMP(*)
!                                 SPECIFICATIONS FOR PARAMETERS
      REAL       TDELTA, U0, U1
      PARAMETER  (TDELTA=0.09, U0=1.0, U1=0.1)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    IWK(2), NDATA
      REAL       DFDATA(2), FDATA(2), XDATA(2)
!                                 SPECIFICATIONS FOR SAVE VARIABLES
      REAL       BREAK(2), CSCOEF(4,2)
      LOGICAL    FIRST
      SAVE       BREAK, CSCOEF, FIRST
!                                 SPECIFICATIONS FOR SUBROUTINES
      DATA FIRST/.TRUE./
!
      IF (FIRST) GO TO 20
   10 CONTINUE
!
!
!                                 Define the boundary conditions
      IF (X .EQ. 0.0) THEN
!                                 These are for x=0.
         ALPHA(1) = 1.0
         BTA(1)  = 0.0
         GAMP(1)  = 0.
!                                 If in the boundary layer,
!                                 compute nonzero gamma prime.
         IF (T .LE. TDELTA) GAMP(1) = CSDER(1,T,BREAK,CSCOEF)
      ELSE
!                                 These are for x=1.
         ALPHA(1) = 0.0
         BTA(1)  = 1.0
         GAMP(1)  = 0.0
      END IF
      RETURN
   20 CONTINUE
!                                 Compute the boundary layer data.
      NDATA     = 2
      XDATA(1)  = 0.0
      XDATA(2)  = TDELTA
      FDATA(1)  = U0
      FDATA(2)  = U1
      DFDATA(1) = 0.0
      DFDATA(2) = 0.0
!                                 Do Hermite cubic interpolation.
      CALL C2HER (NDATA, XDATA, FDATA, DFDATA, BREAK, CSCOEF, IWK)
      FIRST = .FALSE.
      GO TO 10
      END
```

Output

```
                    Solution at T =0.01
    1       2       3       4       5       6       7       8
0.969   0.997   1.000   1.000   1.000   1.000   1.000   1.000

                    Solution at T =0.04
    1       2       3       4       5       6       7       8
0.625   0.871   0.963   0.991   0.998   1.000   1.000   1.000

                    Solution at T =0.09
     1        2        3        4        5        6        7        8
0.0998   0.4603   0.7171   0.8673   0.9437   0.9781   0.9917   0.9951

                    Solution at T =0.16
     1        2        3        4        5        6        7        8
0.0994   0.3127   0.5069   0.6680   0.7893   0.8708   0.9168   0.9316

                    Solution at T =0.25
     1        2        3        4        5        6        7        8
0.0994   0.2564   0.4043   0.5352   0.6428   0.7223   0.7709   0.7873

                    Solution at T =0.36
     1        2        3        4        5        6        7        8
0.0994   0.2172   0.3289   0.4289   0.5123   0.5749   0.6137   0.6268

                    Solution at T =0.49
     1        2        3        4        5        6        7        8
0.0994   0.1847   0.2657   0.3383   0.3989   0.4445   0.4728   0.4824

                    Solution at T =0.64
     1        2        3        4        5        6        7        8
0.0994   0.1583   0.2143   0.2644   0.3063   0.3379   0.3574   0.3641

                    Solution at T =0.81
     1        2        3        4        5        6        7        8
0.0994   0.1382   0.1750   0.2080   0.2356   0.2563   0.2692   0.2736

                    Solution at T =1.00
     1        2        3        4        5        6        7        8
0.0994   0.1237   0.1468   0.1674   0.1847   0.1977   0.2058   0.2085
```

Comments

1. Workspace may be explicitly provided, if desired, by use of M2LCH/DM2LCH. The reference is:

   ```
   CALL M2LCH (IDO, FCNUT, FCNBC, NPDES, T, TEND, NX, XBREAK, TOL,
   HINIT, Y, LDY, WK, IWK)
   ```

   The additional arguments are as follows:

WK — Work array of length `2NX * NPDES(12 * NPDES`^{2} `+ 21 * NPDES + 9).` `WK` should not be changed between calls to `M2LCH`.

IWK — Work array of length `2NX * NPDES. IWK` should not be changed between calls to `M2LCH`.

2. Informational errors

Type	Code	
4	1	After some initial success, the integration was halted by repeated error test failures.
4	2	On the next step, `X` + `H` will equal `X`. Either `TOL` is too small or the problem is stiff.
4	3	After some initial success, the integration was halted by a test on `TOL`.
4	4	Integration was halted after failing to pass the error test even after reducing the step size by a factor of `1.0E` + 10. `TOL` may be too small.
4	5	Integration was halted after failing to achieve corrector convergence even after reducing the step size by a factor of `1.0E` + 10. `TOL` may be too small.

3. Optional usage with Chapter 10 Option Manager

11 This option consists of the parameter `PARAM`, an array with 50 components. See `IVPAG` (page 854) for a more complete documentation of the contents of this array. To reset this option, use the subprogram `SUMAG` for single precision, and `DUMAG` (see Chapter 11, Utilities) for double precision. The entry `PARAM`(1) is assigned the initial step, `HINIT`. The entries `PARAM`(15) and `PARAM`(16) are assigned the values equal to the number of lower and upper diagonals that will occur in the Newton method for solving the BDF corrector equations. The value `PARAM`(17) = 1 is used to signal that the *x* derivatives of the initial data are provided in the the array `Y`. The output values `PARAM`(31)-`PARAM`(36) , showing technical data about the ODE integration, are available with another option manager subroutine call. This call is made after the storage for `MOLCH` is released. The default values for the first 20 entries of `PARAM` are (0, 0, `amach`(2), 500., 0., 5., 0, 0, 1., 3., 1., 2., 2., 1., `amach`(6), `amach`(6), 0, sqrt(`amach`(4)), 1., 0.). Entries 21–50 are defaulted to `amach`(6).

Description

Let M = `NPDES`, N = `NX` and x_i = `XBREAK(I)`. The routine `MOLCH` uses the method of lines to solve the partial differential equation system

$$\frac{\partial u_k}{\partial t} = f_k\left(x, t, u_1, \ldots u_M, \frac{\partial u_1}{\partial x}, \ldots \frac{\partial u_M}{\partial x}, \frac{\partial^2 u_1}{\partial x^2}, \ldots \frac{\partial^2 u_M}{\partial x^2}\right)$$

with the initial conditions

$$u_k = u_k(x, t) \qquad \text{at } t = t_0$$

and the boundary conditions

$$\alpha_k u_k + \beta_k \frac{\partial u_k}{\partial x} = \gamma_k(t) \qquad \text{at } x = x_1 \text{ and at } x = x_N$$

for $k = 1, \ldots, M$.

Cubic Hermite polynomials are used in the x variable approximation so that the trial solution is expanded in the series

$$\hat{u}_k(x, t) = \sum_{i=1}^{N} \left(a_{i,k}(t) \phi_i(x) + b_{i,k}(t) \psi_i(x) \right)$$

where $\phi_i(x)$ and $\psi_i(x)$ are the standard basis functions for the cubic Hermite polynomials with the knots $x_1 < x_2 < \ldots < x_N$. These are piecewise cubic polynomials with continuous first derivatives. At the breakpoints, they satisfy

$$\phi_i(x_l) = \delta_{il} \, \psi_i(x_l) = 0$$
$$\frac{d\phi_i}{dx}(x_l) = 0 \qquad \frac{d\psi_i}{dx}(x_l) = \delta_{il}$$

According to the collocation method, the coefficients of the approximation are obtained so that the trial solution satisfies the differential equation at the two Gaussian points in each subinterval,

$$p_{2j-1} = x_j + \frac{3 - \sqrt{3}}{6}\left(x_{j+1} - x_j\right)$$
$$p_{2j} \quad = x_j + \frac{3 + \sqrt{3}}{6}\left(x_{j+1} + x_j\right)$$

for $j = 1, \ldots, N$. The collocation approximation to the differential equation is

$$\sum_{i=1}^{N} \frac{da_{i,k}}{dt} \phi_i(p_j) + \frac{db_{i,k}}{dt} \psi_i(p_j) =$$
$$f_k\left(p_j, t, \hat{u}_1(p_j), \ldots, \hat{u}_M(p_j), \ldots, (\hat{u}_1)_{xx}(p_j), \ldots, (\hat{u}_M)_{xx}(p_j)\right)$$

for $k = 1, \ldots, M$ and $j = 1, \ldots, 2(N - 1)$.

This is a system of $2M(N - 1)$ ordinary differential equations in $2M\,N$ unknown coefficient functions, $a_{i,k}$ and $b_{i,k}$. This system can be written in the matrix–vector form as $A\ dc/dt = F(t, y)$ with $c(t_0) = c_0$ where c is a vector of coefficients of length $2M\,N$ and c_0 holds the initial values of the coefficients. The last $2M$ equations are obtained by differentiating the boundary conditions

$$\alpha_k \frac{da_k}{dt} + \beta_k \frac{db_k}{dt} = \frac{d\gamma_k}{dt}$$

for $k = 1, \ldots, M$.

The initial conditions $u_k(x, t_0)$ must satisfy the boundary conditions. Also, the $\gamma_k(t)$ must be continuous and have a smooth derivative, or the boundary conditions will not be properly imposed for $t > t_0$.

If $\alpha_k = \beta_k = 0$, it is assumed that no boundary condition is desired for the k-th unknown at the left endpoint. A similar comment holds for the right endpoint. Thus, collocation is done at the endpoint. This is generally a useful feature for systems of first-order partial differential equations.

If the number of partial differential equations is $M = 1$ and the number of breakpoints is $N = 4$, then

$$A = \begin{bmatrix} \alpha_1 & \beta_1 & & & & & & \\ \phi_1(p_1) & \psi_1(p_1) & \phi_2(p_1) & \psi_2(p_1) & & & & \\ \phi_1(p_2) & \psi_1(p_2) & \phi_2(p_2) & \psi_2(p_2) & & & & \\ & & \phi_3(p_3) & \psi_3(p_3) & \phi_4(p_3) & \psi_4(p_3) & & \\ & & \phi_3(p_4) & \psi_3(p_4) & \phi_4(p_4) & \psi_4(p_4) & & \\ & & & & \phi_5(p_5) & \psi_5(p_5) & \phi_6(p_5) & \psi_6(p_5) \\ & & & & \phi_5(p_6) & \psi_5(p_6) & \phi_6(p_6) & \psi_6(p_6) \\ & & & & & & \alpha_4 & \beta_4 \end{bmatrix}$$

The vector c is

$$c = [a_1, b_1, a_2, b_2, a_3, b_3, a_4, b_4]^T$$

and the right-side F is

$$F = \left[\gamma'(x_1), f(p_1), f(p_2), f(p_3), f(p_4), f(p_5), f(p_6), \gamma'(x_4)\right]^T$$

If $M > 1$, then each entry in the above matrix is replaced by an $M \times M$ diagonal matrix. The element α_1 is replaced by diag($\alpha_{1,1}, \ldots, \alpha_{1,\text{M}}$). The elements α_N, β_1 and β_N are handled in the same manner. The $\phi_i(p_j)$ and $\psi_i(p_j)$ elements are replaced by $\phi_i(p_j)I_M$ and $\psi_i(p_j)I_M$ where I_M is the identity matrix of order M. See Madsen and Sincovec (1979) for further details about discretization errors and Jacobian matrix structure.

The input/output array `Y` contains the values of the $a_{k,i}$. The initial values of the $b_{k,i}$ are obtained by using the IMSL cubic spline routine `CSINT` (see Chapter 3, Interpolation and Approximation) to construct functions

$$\hat{u}_k(x, t_0)$$

such that

$$\hat{u}_k(x_i, t_0) = a_{ki}$$

The IMSL routine `CSDER`, see Chapter 3, Interpolation and Approximation, is used to approximate the values

$$\frac{d\hat{U}_k}{dx}\left(x_i, t_0\right) \equiv b_{k,i}$$

There is an optional usage of MOLCH that allows the user to provide the initial values of $b_{k,i}$.

The order of matrix A is $2M\,N$ and its maximum bandwidth is $6M - 1$. The band structure of the Jacobian of F with respect to c is the same as the band structure of A. This system is solved using a modified version of IVPAG, page 854. Some of the linear solvers were removed. Numerical Jacobians are used exclusively. The algorithm is unchanged. Gear's BDF method is used as the default because the system is typically stiff.

We now present four examples of PDEs that illustrate how users can interface their problems with IMSL PDE solving software. The examples are small and not indicative of the complexities that most practitioners will face in their applications. A set of seven sample application problems, some of them with more than one equation, is given in Sincovec and Madsen (1975). Two further examples are given in Madsen and Sincovec (1979).

Additonal Examples

Example 2

In this example, using MOLCH, we solve the linear normalized diffusion PDE $u_t = u_{xx}$ but with an optional usage that provides values of the derivatives, u_x, of the initial data. Due to errors in the numerical derivatives computed by spline interpolation, more precise derivative values are required when the initial data is $u(x, 0) = 1 + \cos[(2n - 1)\pi x]$, $n > 1$. The boundary conditions are "zero flux" conditions $u_x(0, t) = u_x(1, t) = 0$ for $t > 0$. Note that the initial data is compatible with these end conditions since the derivative function

$$u_x\left(x,0\right) = \frac{du\left(x,0\right)}{dx} = -\left(2n-1\right)\pi \sin\left[\left(2n-1\right)\pi x\right]$$

vanishes at $x = 0$ and $x = 1$.

The example illustrates the use of the IMSL options manager subprograms SUMAG or, for double precision, DUMAG, see Chapter 11, Utilities, to reset the array PARAM used for control of the specialized version of IVPAG that integrates the system of ODEs. This optional usage signals that the derivative of the initial data is passed by the user. The values $u(x, tend)$ and $u_x(x, tend)$ are output at the breakpoints with the optional usage.

```
      USE IMSL_LIBRARIES
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    LDY, NPDES, NX
      PARAMETER  (NPDES=1, NX=10, LDY=NPDES)
!                                 SPECIFICATIONS FOR PARAMETERS
      INTEGER    ICHAP, IGET, IPUT, KPARAM
      PARAMETER  (ICHAP=5, IGET=1, IPUT=2, KPARAM=11)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    I, IACT, IDO, IOPT(1), J, JGO, N, NOUT, NSTEP
      REAL       ARG1, HINIT, PREC, PARAM(50), PI, T, TEND, TOL, &
                 XBREAK(NX), Y(LDY,2*NX)
      CHARACTER  TITLE*36
!                                 SPECIFICATIONS FOR INTRINSICS
```

```
      INTRINSIC  COS, FLOAT, SIN, SQRT
      REAL       COS, FLOAT, SIN, SQRT
!                                 SPECIFICATIONS FOR FUNCTIONS
      EXTERNAL   FCNBC, FCNUT
!                                 Set breakpoints and initial
!                                 conditions.
      N       = 5
      PI      = CONST(’pi’)
      IOPT(1) = KPARAM
      DO 10  I=1, NX
         XBREAK(I) = FLOAT(I-1)/(NX-1)
         ARG1       = (2.*N-1)*PI
!                                 Set function values.
         Y(1,I) = 1. + COS(ARG1*XBREAK(I))
!                                 Set first derivative values.
         Y(1,I+NX) = -ARG1*SIN(ARG1*XBREAK(I))
   10 CONTINUE
!                                 Set parameters for MOLCH
      PREC = AMACH(4)
      TOL   = SQRT(PREC)
      HINIT = 0.01*TOL
      T     = 0.0
      IDO   = 1
      NSTEP = 10
      CALL UMACH (2, NOUT)
      J = 0
!                                  Get and reset the PARAM array
!                                  so that user-provided derivatives
!                                  of the initial data are used.
      JGO  = 1
      IACT = IGET
      GO TO 70
   20 CONTINUE
!                                  This flag signals that
!                                  derivatives are passed.
      PARAM(17) = 1.
      JGO       = 2
      IACT      = IPUT
      GO TO 70
   30 CONTINUE
!                                  Look at output at steps
!                                  of 0.001.
      TEND = 0.
   40 CONTINUE
      J    = J + 1
      TEND = TEND + 0.001
!                                 Solve the problem
      CALL MOLCH (IDO, FCNUT, FCNBC, T, TEND,  XBREAK, Y, NPDES=NPDES, &
                  NX=NX, HINIT=HINIT, TOL=TOL)
      IF (J .LE. NSTEP) THEN
!                                 Print results
         WRITE (TITLE,’(A,F5.3)’) ’Solution and derivatives at T =’, T
         CALL WRRRN (TITLE, Y)
!                                 Final call to release workspace
         IF (J .EQ. NSTEP) IDO = 3
         GO TO 40
      END IF
!                                  Show, for example, the maximum
!                                  step size used.
      JGO  = 3
      IACT = IGET
      GO TO 70
   50 CONTINUE
      WRITE (NOUT,*) ’ Maximum step size used is:  ’, PARAM(33)
!                                  Reset option to defaults
      JGO     = 4
      IAC     = IPUT
      IOPT(1) = -IOPT(1)
      GO TO 70
   60 CONTINUE
      RETURN
!                                  Internal routine to work options
   70 CONTINUE
      CALL SUMAG (’math’, ICHAP, IACT, IOPT, PARAM, numopt=1)
      GO TO (20, 30, 50, 60), JGO
      END
      SUBROUTINE FCNUT (NPDES, X, T, U, UX, UXX, UT)
!                                  SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NPDES
      REAL       X, T, U(*), UX(*), UXX(*), UT(*)
!
!                                  Define the PDE
      UT(1) = UXX(1)
      RETURN
      END
      SUBROUTINE FCNBC (NPDES, X, T, ALPHA, BTA, GAMP)
!                                  SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NPDES
      REAL       X, T, ALPHA(*), BTA(*), GAMP(*)
!
      ALPHA(1) = 0.0
      BTA(1)   = 1.0
      GAMP(1)  = 0.0
      RETURN
      END
```

Output

```
                Solution and derivatives at T =0.001
     1      2       3      4       5      6       7      8       9     10
 1.483  0.517   1.483  0.517   1.483  0.517   1.483  0.517   1.483  0.517

    11     12      13     14      15     16      17     18      19     20
 0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000

                Solution and derivatives at T =0.002
     1      2       3      4       5      6       7      8       9     10
 1.233  0.767   1.233  0.767   1.233  0.767   1.233  0.767   1.233  0.767

    11     12      13     14      15     16      17     18      19     20
```

```
0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000

               Solution and derivatives at T =0.003
    1      2       3      4       5      6       7      8       9     10
1.113  0.887   1.113  0.887   1.113  0.887   1.113  0.887   1.113  0.887

   11     12      13     14      15     16      17     18      19     20
0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000

               Solution and derivatives at T =0.004
    1      2       3      4       5      6       7      8       9     10
1.054  0.946   1.054  0.946   1.054  0.946   1.054  0.946   1.054  0.946

   11     12      13     14      15     16      17     18      19     20
0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000

               Solution and derivatives at T =0.005
    1      2       3      4       5      6       7      8       9     10
1.026  0.974   1.026  0.974   1.026  0.974   1.026  0.974   1.026  0.974

   11     12      13     14      15     16      17     18      19     20
0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000

               Solution and derivatives at T =0.006
    1      2       3      4       5      6       7      8       9     10
1.012  0.988   1.012  0.988   1.012  0.988   1.012  0.988   1.012  0.988

   11     12      13     14      15     16      17     18      19     20
0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000

               Solution and derivatives at T =0.007
    1      2       3      4       5      6       7      8       9     10
1.006  0.994   1.006  0.994   1.006  0.994   1.006  0.994   1.006  0.994

   11     12      13     14      15     16      17     18      19     20
0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000

               Solution and derivatives at T =0.008
    1      2       3      4       5      6       7      8       9     10
1.003  0.997   1.003  0.997   1.003  0.997   1.003  0.997   1.003  0.997

   11     12      13     14      15     16      17     18      19     20
0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000


               Solution and derivatives at T =0.009
    1      2       3      4       5      6       7      8       9     10
1.001  0.999   1.001  0.999   1.001  0.999   1.001  0.999   1.001  0.999

   11     12      13     14      15     16      17     18      19     20
0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000

               Solution and derivatives at T =0.010
    1      2       3      4       5      6       7      8       9     10
1.001  0.999   1.001  0.999   1.001  0.999   1.001  0.999   1.001  0.999
```

```
    11     12      13     14      15     16      17     18      19     20
 0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000   0.000  0.000
Maximum step size used is:      1.00000E-02
```

Example 3

In this example, we consider the linear normalized hyperbolic PDE, $u_{tt} = u_{xx}$, the “vibrating string” equation. This naturally leads to a system of first order PDEs. Define a new dependent variable $u_t = v$. Then, $v_t = u_{xx}$ is the second equation in the system. We take as initial data $u(x, 0) = \sin(\pi x)$ and $u_t(x, 0) = v(x, 0) = 0$. The ends of the string are fixed so $u(0, t) = u(1, t) = v(0, t) = v(1, t) = 0$. The exact solution to this problem is $u(x, t) = \sin(\pi x) \cos(\pi t)$. Residuals are computed at the output values of t for $0 < t \le 2$. Output is obtained at 200 steps in increments of 0.01.

Even though the sample code `MOLCH` gives satisfactory results for this PDE, users should be aware that for *nonlinear problems*, “shocks” can develop in the solution. The appearance of shocks may cause the code to fail in unpredictable ways. See Courant and Hilbert (1962), pages 488-490, for an introductory discussion of shocks in hyperbolic systems.

```
      USE IMSL_LIBRARIES
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    LDY, NPDES, NX
      PARAMETER  (NPDES=2, NX=10, LDY=NPDES)
!                                 SPECIFICATIONS FOR PARAMETERS
      INTEGER    ICHAP, IGET, IPUT, KPARAM
      PARAMETER  (ICHAP=5, IGET=1, IPUT=2, KPARAM=11)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    I, IACT, IDO, IOPT(1), J, JGO, NOUT, NSTEP
      REAL       HINIT, PREC, PARAM(50), PI, T, TEND, TOL, XBREAK(NX), &
                Y(LDY,2*NX), ERROR(NX)
!                                 SPECIFICATIONS FOR INTRINSICS
      INTRINSIC  COS, FLOAT, SIN, SQRT
      REAL       COS, FLOAT, SIN, SQRT
!                                 SPECIFICATIONS FOR SUBROUTINES
!                                 SPECIFICATIONS FOR FUNCTIONS
      EXTERNAL   FCNBC, FCNUT
!                                 Set breakpoints and initial
!                                 conditions.
      PI      = CONST(’pi’)
      IOPT(1) = KPARAM
      DO 10  I=1, NX
         XBREAK(I) = FLOAT(I-1)/(NX-1)
!                                 Set function values.
         Y(1,I) = SIN(PI*XBREAK(I))
         Y(2,I) = 0.
!                                 Set first derivative values.
         Y(1,I+NX) = PI*COS(PI*XBREAK(I))
         Y(2,I+NX) = 0.0
   10 CONTINUE
!                                 Set parameters for MOLCH
      PREC = AMACH(4)
      TOL   = 0.1*SQRT(PREC)
      HINIT = 0.01*TOL
      T     = 0.0
      IDO   = 1
      NSTEP = 200
      CALL UMACH (2, NOUT)
      J = 0
!                                  Get and reset the PARAM array
!                                  so that user-provided derivatives
!                                  of the initial data are used.
      JGO  = 1
      IACT = IGET
      GO TO 90
   20 CONTINUE
!                                  This flag signals that
!                                  derivatives are passed.
      PARAM(17) = 1.
      JGO       = 2
      IACT      = IPUT
      GO TO 90
   30 CONTINUE
!                                  Look at output at steps
!                                  of 0.01 and compute errors.
      ERRU = 0.
      TEND = 0.
   40 CONTINUE
      J    = J + 1
      TEND = TEND + 0.01
!                                 Solve the problem
      CALL MOLCH (IDO, FCNUT, FCNBC, T, TEND, XBREAK, Y, NX=NX, &
                  HINIT=HINIT, TOL=TOL)
      DO 50  I=1, NX
         ERROR(I) = Y(1,I) - SIN(PI*XBREAK(I))*COS(PI*TEND)
   50 CONTINUE
      IF (J .LE. NSTEP) THEN
         DO 60  I=1, NX
            ERRU = AMAX1(ERRU,ABS(ERROR(I)))
   60    CONTINUE
!                                 Final call to release workspace
         IF (J .EQ. NSTEP) IDO = 3
         GO TO 40
      END IF
!                                 Show, for example, the maximum
!                                 step size used.
      JGO  = 3
      IACT = IGET
      GO TO 90
   70 CONTINUE
      WRITE (NOUT,*) ’ Maximum error in u(x,t) divided by TOL: ’, &
                   ERRU/TOL
      WRITE (NOUT,*) ’ Maximum step size used is:  ’, PARAM(33)
!                                 Reset option to defaults
      JGO     = 4
      IACT    = IPUT
      IOPT(1) = -IOPT(1)
      GO TO 90
   80 CONTINUE
```

```
      RETURN
!                                  Internal routine to work options
   90 CONTINUE
      CALL SUMAG ('math', ICHAP, IACT, IOPT, PARAM)
      GO TO (20, 30, 70, 80), JGO
      END
      SUBROUTINE FCNUT (NPDES, X, T, U, UX, UXX, UT)
!                                  SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NPDES
      REAL       X, T, U(*), UX(*), UXX(*), UT(*)
!
!                                  Define the PDE
      UT(1) = U(2)
      UT(2) = UXX(1)
      RETURN
      END
      SUBROUTINE FCNBC (NPDES, X, T, ALPHA, BTA, GAMP)
!                                  SPECIFICATIONS FOR ARGUMENTS
      INTEGER    NPDES
      REAL       X, T, ALPHA(*), BTA(*), GAMP(*)
!
      ALPHA(1) = 1.0
      BTA(1)  = 0.0
      GAMP(1)  = 0.0
      ALPHA(2) = 1.0
      BTA(2)  = 0.0
      GAMP(2)  = 0.0
      RETURN
      END
```

Output

```
Maximum error in u(x,t) divided by TOL:     1.28094
Maximum step size used is:      9.99999E-02
```

FPS2H

Solves Poisson's or Helmholtz's equation on a two-dimensional rectangle using a fast Poisson solver based on the HODIE finite-difference scheme on a uniform mesh.

Required Arguments

PRHS — User-supplied FUNCTION to evaluate the right side of the partial differential equation. The form is PRHS(X, Y), where

X – X-coordinate value.   (Input)
Y – Y-coordinate value.   (Input)
PRHS – Value of the right side at (X, Y).   (Output)

PRHS must be declared EXTERNAL in the calling program.

BRHS — User-supplied FUNCTION to evaluate the right side of the boundary conditions. The form is BRHS(ISIDE, X, Y), where

ISIDE – Side number. (Input)
See IBCTY below for the definition of the side numbers.
X – X-coordinate value. (Input)
Y – Y-coordinate value. (Input)
BRHS – Value of the right side of the boundary condition at (X, Y). (Output)
BRHS must be declared EXTERNAL in the calling program.

COEFU — Value of the coefficient of U in the differential equation. (Input)

NX — Number of grid lines in the X-direction. (Input)
NX must be at least 4. See Comment 2 for further restrictions on NX.

NY — Number of grid lines in the Y-direction. (Input)
NY must be at least 4. See Comment 2 for further restrictions on NY.

AX — The value of X along the left side of the domain. (Input)

BX — The value of X along the right side of the domain. (Input)

AY — The value of Y along the bottom of the domain. (Input)

BY — The value of Y along the top of the domain. (Input)

IBCTY — Array of size 4 indicating the type of boundary condition on each side of the domain or that the solution is periodic. (Input)
The sides are numbered 1 to 4 as follows:

Side	**Location**
1 - Right	(X = BX)
2 - Bottom	(Y = AY)
3 - Left	(X = AX)
4 - Top	(Y = BY)

There are three boundary condition types.

IBCTY	**Boundary Condition**
1	Value of U is given. (Dirichlet)

2 Value of dU/dX is given (sides 1 and/or 3). (Neumann) Value of dU/dY is given (sides 2 and/or 4).

3 Periodic.

U — Array of size `NX` by `NY` containing the solution at the grid points. (Output)

Optional Arguments

IORDER — Order of accuracy of the finite-difference approximation. (Input)
It can be either 2 or 4. Usually, `IORDER` = 4 is used.
Default: `IORDER` = 4.

LDU — Leading dimension of `U` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDU` = size (`U`,1).

FORTRAN 90 Interface

Generic:
```
CALL FPS2H (PRHS, BRHS, COEFU, NX, NY, AX, BX, AY, BY,
IBCTY, U [,…])
```

Specific: The specific interface names are `S_FPS2H` and `D_FPS2H`.

FORTRAN 77 Interface

Single:
```
CALL FPS2H (PRHS, BRHS, COEFU, NX, NY, AX, BX, AY, BY,
IBCTY, IORDER, U, LDU)
```

Double: The double precision name is `DFPS2H`.

Example

In this example, the equation

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+3u=-2\sin\left(x+2y\right)+16e^{2x+3y}$$

with the boundary conditions $\partial u/\partial y = 2\cos(x + 2y) + 3\exp(2x + 3y)$ on the bottom side and $u = \sin(x + 2y) + \exp(2x + 3y)$ on the other three sides. The domain is the rectangle[0, 1/4] × [0, 1/2]. The output of `FPS2H` is a 17 × 33 table of *U* values. The quadratic interpolation routine `QD2VL` is used to print a table of values.

```
 USE FPS2H_INT
 USE QD2VL_INT
 USE UMACH_INT
 INTEGER    NCVAL, NX, NXTABL, NY, NYTABL
 PARAMETER  (NCVAL=11, NX=17, NXTABL=5, NY=33, NYTABL=5)
!
```

```
      INTEGER    I, IBCTY(4), IORDER, J, NOUT
      REAL       AX, AY, BRHS, BX, BY, COEFU, ERROR, FLOAT, PRHS, &
                TRUE, U(NX,NY), UTABL, X, XDATA(NX), Y, YDATA(NY)
      INTRINSIC  FLOAT
      EXTERNAL   BRHS, PRHS
!                                 Set rectangle size
      AX = 0.0
      BX = 0.25
      AY = 0.0
      BY = 0.50
!                                 Set boundary condition types
      IBCTY(1) = 1
      IBCTY(2) = 2
      IBCTY(3) = 1
      IBCTY(4) = 1
!                                 Coefficient of U
      COEFU = 3.0
!                                 Order of the method
      IORDER = 4
!                                 Solve the PDE
      CALL FPS2H (PRHS, BRHS, COEFU, NX, NY, AX, BX, AY, BY, IBCTY, U)
!                                 Setup for quadratic interpolation
      DO 10  I=1, NX
         XDATA(I) = AX + (BX-AX)*FLOAT(I-1)/FLOAT(NX-1)
   10 CONTINUE
      DO 20  J=1, NY
         YDATA(J) = AY + (BY-AY)*FLOAT(J-1)/FLOAT(NY-1)
   20 CONTINUE
!                                 Print the solution
      CALL UMACH (2, NOUT)
      WRITE (NOUT,’(8X,A,11X,A,11X,A,8X,A)’) ’X’, ’Y’, ’U’, ’Error’
      DO 40  J=1, NYTABL
         DO 30  I=1, NXTABL
            X     = AX + (BX-AX)*FLOAT(I-1)/FLOAT(NXTABL-1)
            Y     = AY + (BY-AY)*FLOAT(J-1)/FLOAT(NYTABL-1)
            UTABL = QD2VL(X,Y,XDATA,YDATA,U)
            TRUE  = SIN(X+2.*Y) + EXP(2.*X+3.*Y)
            ERROR = TRUE - UTABL
            WRITE (NOUT,’(4F12.4)’) X, Y, UTABL, ERROR
   30 CONTINUE
   40 CONTINUE
      END
!
      REAL FUNCTION PRHS (X, Y)
      REAL       X, Y
!
      REAL       EXP, SIN
      INTRINSIC  EXP, SIN
!                                 Define right side of the PDE
      PRHS = -2.*SIN(X+2.*Y) + 16.*EXP(2.*X+3.*Y)
      RETURN
      END
!
      REAL FUNCTION BRHS (ISIDE, X, Y)
      INTEGER    ISIDE
      REAL       X, Y
!
      REAL       COS, EXP, SIN
      INTRINSIC  COS, EXP, SIN
!                                  Define the boundary conditions
      IF (ISIDE .EQ. 2) THEN
         BRHS = 2.*COS(X+2.*Y) + 3.*EXP(2.*X+3.*Y)
      ELSE
         BRHS = SIN(X+2.*Y) + EXP(2.*X+3.*Y)
      END IF
      RETURN
      END
```

Output

```
    X           Y           U         Error
 0.0000      0.0000      1.0000      0.0000
 0.0625      0.0000      1.1956      0.0000
 0.1250      0.0000      1.4087      0.0000
 0.1875      0.0000      1.6414      0.0000
 0.2500      0.0000      1.8961      0.0000
 0.0000      0.1250      1.7024      0.0000
 0.0625      0.1250      1.9562      0.0000
 0.1250      0.1250      2.2345      0.0000
 0.1875      0.1250      2.5407      0.0000
 0.2500      0.1250      2.8783      0.0000
 0.0000      0.2500      2.5964      0.0000
 0.0625      0.2500      2.9322      0.0000
 0.1250      0.2500      3.3034      0.0000
 0.1875      0.2500      3.7148      0.0000
 0.2500      0.2500      4.1720      0.0000
 0.0000      0.3750      3.7619      0.0000
 0.0625      0.3750      4.2163      0.0000
 0.1250      0.3750      4.7226      0.0000
 0.1875      0.3750      5.2878      0.0000
 0.2500      0.3750      5.9199      0.0000
 0.0000      0.5000      5.3232      0.0000
 0.0625      0.5000      5.9520      0.0000
 0.1250      0.5000      6.6569      0.0000
 0.1875      0.5000      7.4483      0.0000
 0.2500      0.5000      8.3380      0.0000
```

Comments

1. Workspace may be explicitly provided, if desired, by use of F2S2H/DF2S2H. The reference is:

   ```
   CALL F2S2H (PRHS, BRHS, COEFU, NX, NY, AX, BX, AY, BY, IBCTY,
   IORDER, U, LDU, UWORK, WORK)
   ```

   The additional arguments are as follows:

   UWORK — Work array of size NX + 2 by NY + 2. If the actual dimensions of U are large enough, then U and UWORK can be the same array.

WORK — Work array of length `(NX + 1)(NY + 1)(IORDER – 2)/2 + 6(NX + NY) + NX/2 + 16`.

2. The grid spacing is the distance between the (uniformly spaced) grid lines. It is given by the formulas `HX` = (`BX` – `AX`)/(`NX` – 1) and `HY` = (`BY` – `AY`)/(`NY` – 1). The grid spacings in the `X` and `Y` directions must be the same, i.e., `NX` and `NY` must be such that `HX` equals `HY`. Also, as noted above, `NX` and `NY` must both be at least 4. To increase the speed of the fast Fourier transform, `NX` – 1 should be the product of small primes. Good choices are 17, 33, and 65.

3. If –`COEFU` is nearly equal to an eigenvalue of the Laplacian with homogeneous boundary conditions, then the computed solution might have large errors.

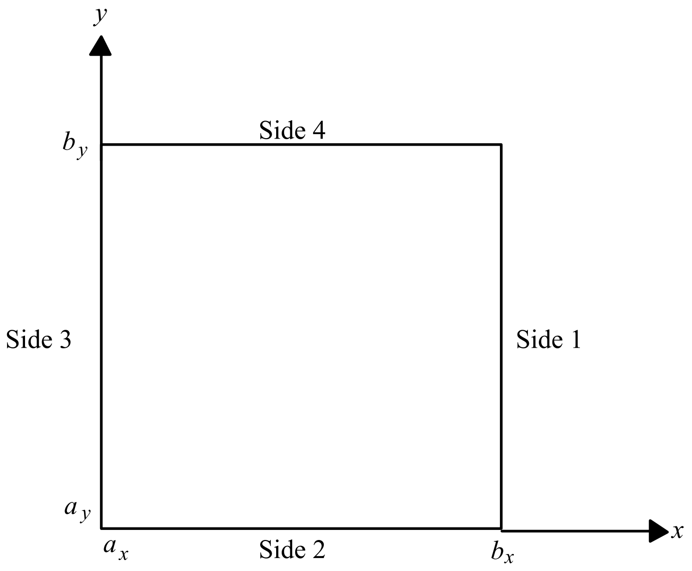
Description

Let c = `COEFU`, a_x = `AX`, b_x = `BX`, a_y = `AY`, b_y = `BY`, n_x = `NX` and n_y = `NY`.

`FPS2H` is based on the code `HFFT2D` by Boisvert (1984). It solves the equation

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+cu=p$$

on the rectangular domain $(a_x, b_x) \times (a_y, b_y)$ with a user-specified combination of Dirichlet (solution prescribed), Neumann (first-derivative prescribed), or periodic boundary conditions. The sides are numbered clockwise, starting with the right side.



When c = 0 and only Neumann or periodic boundary conditions are prescribed, then any constant may be added to the solution to obtain another solution to the problem. In this case, the solution of minimum ∞-norm is returned.

The solution is computed using either a second-or fourth-order accurate finite-difference approximation of the continuous equation. The resulting system of linear algebraic equations is solved using fast Fourier transform techniques. The algorithm relies upon the fact that $n_x - 1$ is highly composite (the product of small primes). For details of the algorithm, see Boisvert (1984). If $n_x - 1$ is highly composite then the execution time of FPS2H is proportional to $n_x n_y \log_2 n_x$. If evaluations of $p(x, y)$ are inexpensive, then the difference in running time between IORDER = 2 and IORDER = 4 is small.

FPS3H

Solves Poisson's or Helmholtz's equation on a three-dimensional box using a fast Poisson solver based on the HODIE finite-difference scheme on a uniform mesh.

Required Arguments

PRHS — User-supplied FUNCTION to evaluate the right side of the partial differential equation. The form is PRHS(X, Y, Z), where

X – The *x*-coordinate value. (Input)
Y – The *y*-coordinate value. (Input)
Z – The *z*-coordinate value. (Input)
PRHS – Value of the right side at (X, Y, Z). (Output)

PRHS must be declared EXTERNAL in the calling program.

BRHS — User-supplied FUNCTION to evaluate the right side of the boundary conditions. The form is BRHS(ISIDE, X, Y, Z), where

ISIDE – Side number. (Input)
See IBCTY for the definition of the side numbers.
X – The *x*-coordinate value. (Input)
Y – The *y*-coordinate value. (Input)
Z – The *z*-coordinate value. (Input)
BRHS – Value of the right side of the boundary condition at (X, Y, Z). (Output)

BRHS must be declared EXTERNAL in the calling program.

COEFU — Value of the coefficient of U in the differential equation. (Input)

NX — Number of grid lines in the *x*-direction. (Input)
NX must be at least 4. See Comment 2 for further restrictions on NX.

NY — Number of grid lines in the *y*-direction. (Input)
NY must be at least 4. See Comment 2 for further restrictions on NY.

NZ — Number of grid lines in the *y*-direction. (Input)
NZ must be at least 4. See Comment 2 for further restrictions on NZ.

AX — Value of X along the left side of the domain. (Input)

BX — Value of X along the right side of the domain. (Input)

AY — Value of Y along the bottom of the domain. (Input)

BY — Value of Y along the top of the domain. (Input)

AZ — Value of Z along the front of the domain. (Input)

BZ — Value of Z along the back of the domain. (Input)

IBCTY — Array of size 6 indicating the type of boundary condition on each face of the domain or that the solution is periodic. (Input)
The sides are numbers 1 to 6 as follows:

Side	**Location**
1 - Right	(X = BX)
2 - Bottom	(Y = AY)
3 - Left	(X = AX)
4 - Top	(Y = BY)
5 - Front	(Z = BZ)
6 - Back	(Z = AZ)

There are three boundary condition types.

IBCTY	**Boundary Condition**
1	Value of U is given. (Dirichlet)
2	Value of dU/dX is given (sides 1 and/or 3). (Neumann) Value of dU/dY is given (sides 2 and/or 4). Value of dU/dZ is given (sides 5 and/or 6).
3	Periodic.

U — Array of size NX by NY by NZ containing the solution at the grid points. (Output)

Optional Arguments

IORDER — Order of accuracy of the finite-difference approximation. (Input)
It can be either 2 or 4. Usually, `IORDER` = 4 is used.
Default: `IORDER` = 4.

LDU — Leading dimension of `U` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDU` = size (`U`,1).

MDU — Middle dimension of `U` exactly as specified in the dimension statement of the calling program. (Input)
Default: `MDU` = size (`U`,2).

FORTRAN 90 Interface

Generic: `CALL FPS3H (PRHS, BRHS, COEFU, NX, NY, NZ, AX, BX, AY, BY, AZ, BZ, IBCTY, U [,…])`

Specific: The specific interface names are `S_FPS3H` and `D_FPS3H`.

FORTRAN 77 Interface

Single: `CALL FPS3H (PRHS, BRHS, COEFU, NX, NY, NZ, AX, BX, AY, BY, AZ, BZ, IBCTY, IORDER, U, LDU, MDU)`

Double: The double precision name is `DFPS3H`.

Example

This example solves the equation

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}+10u=-4\cos\left(3x+y-2z\right)+12e^{x-z}+10$$

with the boundary conditions $\partial u/\partial z = -2\sin(3x + y - 2z) - \exp(x - z)$ on the front side and $u = \cos(3x + y - 2z) + \exp(x - z) + 1$ on the other five sides. The domain is the box [0, 1/4] × [0, 1/2] × [0, 1/2]. The output of `FPS3H` is a 9 × 17 × 17 table of U values. The quadratic interpolation routine `QD3VL` is used to print a table of values.

```
      USE FPS3H_INT
      USE UMACH_INT
      USE QD3VL_INT
!                                  SPECIFICATIONS FOR PARAMETERS
      INTEGER    LDU, MDU, NX, NXTABL, NY, NYTABL, NZ, NZTABL
      PARAMETER  (NX=5, NXTABL=4, NY=9, NYTABL=3, NZ=9, &
                 NZTABL=3, LDU=NX, MDU=NY)
!
      INTEGER    I, IBCTY(6), IORDER, J, K, NOUT
      REAL       AX, AY, AZ, BRHS, BX, BY, BZ, COEFU, FLOAT, PRHS, &
```

```
                 U(LDU,MDU,NZ), UTABL, X, ERROR, TRUE, &
                 XDATA(NX), Y, YDATA(NY), Z, ZDATA(NZ)
      INTRINSIC  COS, EXP, FLOAT
      EXTERNAL   BRHS, PRHS
!                                 Define domain
      AX = 0.0
      BX = 0.125
      AY = 0.0
      BY = 0.25
      AZ = 0.0
      BZ = 0.25
!                                 Set boundary condition types
      IBCTY(1) = 1
      IBCTY(2) = 1
      IBCTY(3) = 1
      IBCTY(4) = 1
      IBCTY(5) = 2
      IBCTY(6) = 1
!                                 Coefficient of U
      COEFU = 10.0
!                                 Order of the method
      IORDER = 4
!                                 Solve the PDE
      CALL FPS3H (PRHS, BRHS, COEFU, NX, NY, NZ, AX, BX, AY, BY, AZ, &
                 BZ, IBCTY, U)
!                                 Set up for quadratic interpolation
      DO 10  I=1, NX
         XDATA(I) = AX + (BX-AX)*FLOAT(I-1)/FLOAT(NX-1)
   10 CONTINUE
      DO 20  J=1, NY
         YDATA(J) = AY + (BY-AY)*FLOAT(J-1)/FLOAT(NY-1)
   20 CONTINUE
      DO 30  K=1, NZ
         ZDATA(K) = AZ + (BZ-AZ)*FLOAT(K-1)/FLOAT(NZ-1)
   30 CONTINUE
!                                 Print the solution
      CALL UMACH (2, NOUT)
      WRITE (NOUT,'(8X,5(A,11X))') 'X', 'Y', 'Z', 'U', 'Error'
      DO 60  K=1, NZTABL
         DO 50  J=1, NYTABL
            DO 40  I=1, NXTABL
               X     = AX + (BX-AX)*FLOAT(I-1)/FLOAT(NXTABL-1)
               Y     = AY + (BY-AY)*FLOAT(J-1)/FLOAT(NYTABL-1)
               Z     = AZ + (BZ-AZ)*FLOAT(K-1)/FLOAT(NZTABL-1)
               UTABL = QD3VL(X,Y,Z,XDATA,YDATA,ZDATA,U, CHECK=.false.)
               TRUE = COS(3.0*X+Y-2.0*Z) + EXP(X-Z) + 1.0
               ERROR = UTABL - TRUE
               WRITE (NOUT,'(5F12.4)') X, Y, Z, UTABL, ERROR
   40       CONTINUE
   50    CONTINUE
   60 CONTINUE
      END
!
      REAL FUNCTION PRHS (X, Y, Z)
      REAL       X, Y, Z
```

```
!
      REAL       COS, EXP
      INTRINSIC  COS, EXP
!                                 Right side of the PDE
      PRHS = -4.0*COS(3.0*X+Y-2.0*Z) + 12*EXP(X-Z) + 10.0
      RETURN
      END
!
      REAL FUNCTION BRHS (ISIDE, X, Y, Z)
      INTEGER    ISIDE
      REAL       X, Y, Z
!
      REAL       COS, EXP, SIN
      INTRINSIC  COS, EXP, SIN
!                                 Boundary conditions
      IF (ISIDE .EQ. 5) THEN
         BRHS = -2.0*SIN(3.0*X+Y-2.0*Z) - EXP(X-Z)
      ELSE
         BRHS = COS(3.0*X+Y-2.0*Z) + EXP(X-Z) + 1.0
      END IF
      RETURN
      END
```

Output

```
   X           Y           Z           U           Error
 0.0000      0.0000      0.0000      3.0000      0.0000
 0.0417      0.0000      0.0000      3.0348      0.0000
 0.0833      0.0000      0.0000      3.0558      0.0001
 0.1250      0.0000      0.0000      3.0637      0.0001
 0.0000      0.1250      0.0000      2.9922      0.0000
 0.0417      0.1250      0.0000      3.0115      0.0000
 0.0833      0.1250      0.0000      3.0175      0.0000
 0.1250      0.1250      0.0000      3.0107      0.0000
 0.0000      0.2500      0.0000      2.9690      0.0001
 0.0417      0.2500      0.0000      2.9731      0.0000
 0.0833      0.2500      0.0000      2.9645      0.0000
 0.1250      0.2500      0.0000      2.9440     -0.0001
 0.0000      0.0000      0.1250      2.8514      0.0000
 0.0417      0.0000      0.1250      2.9123      0.0000
 0.0833      0.0000      0.1250      2.9592      0.0000
 0.1250      0.0000      0.1250      2.9922      0.0000
 0.0000      0.1250      0.1250      2.8747      0.0000
 0.0417      0.1250      0.1250      2.9211      0.0010
 0.0833      0.1250      0.1250      2.9524      0.0010
 0.1250      0.1250      0.1250      2.9689      0.0000
 0.0000      0.2500      0.1250      2.8825      0.0000
 0.0417      0.2500      0.1250      2.9123      0.0000
 0.0833      0.2500      0.1250      2.9281      0.0000
 0.1250      0.2500      0.1250      2.9305      0.0000
 0.0000      0.0000      0.2500      2.6314     -0.0249
 0.0417      0.0000      0.2500      2.7420     -0.0004
 0.0833      0.0000      0.2500      2.8112     -0.0042
 0.1250      0.0000      0.2500      2.8609     -0.0138
 0.0000      0.1250      0.2500      2.7093      0.0000
```

```
0.0417      0.1250      0.2500      2.8153       0.0344
0.0833      0.1250      0.2500      2.8628       0.0237
0.1250      0.1250      0.2500      2.8825       0.0000
0.0000      0.2500      0.2500      2.7351      -0.0127
0.0417      0.2500      0.2500      2.8030      -0.0011
0.0833      0.2500      0.2500      2.8424      -0.0040
0.1250      0.2500      0.2500      2.8735      -0.0012
```

Comments

1. Workspace may be explicitly provided, if desired, by use of F2S3H/DF2S3H. The reference is:

   ```
   CALL F2S3H (PRHS, BRHS, COEFU, NX, NY, NZ, AX, BX,
   AY, BY, AZ, BZ, IBCTY, IORDER, U, LDU,
   MDU, UWORK, WORK)
   ```

   The additional arguments are as follows:

   UWORK — Work array of size NX + 2 by NY + 2 by NZ + 2. If the actual dimensions of U are large enough, then U and UWORK can be the same array.

   WORK — Work array of length (NX + 1)(NY + 1)(NZ + 1)(IORDER – 2)/2 + 2(NX * NY + NX * NZ + NY * NZ) + 2(NX + NY + 1) + MAX(2 * NX * NY, 2 * NX + NY + 4 * NZ + (NX + NZ)/2 + 29)

2. The grid spacing is the distance between the (uniformly spaced) grid lines. It is given by the formulas
   HX = (BX – AX)/(NX – 1),
   HY = (BY – AY)/(NY – 1), and
   HZ = (BZ – AZ)/(NZ – 1).
   The grid spacings in the X, Y and Z directions must be the same, i.e., NX, NY and NZ must be such that HX = HY = HZ. Also, as noted above, NX, NY and NZ must all be at least 4. To increase the speed of the Fast Fourier transform, NX – 1 and NZ – 1 should be the product of small primes. Good choices for NX and NZ are 17, 33 and 65.

3. If −COEFU is nearly equal to an eigenvalue of the Laplacian with homogeneous boundary conditions, then the computed solution might have large errors.

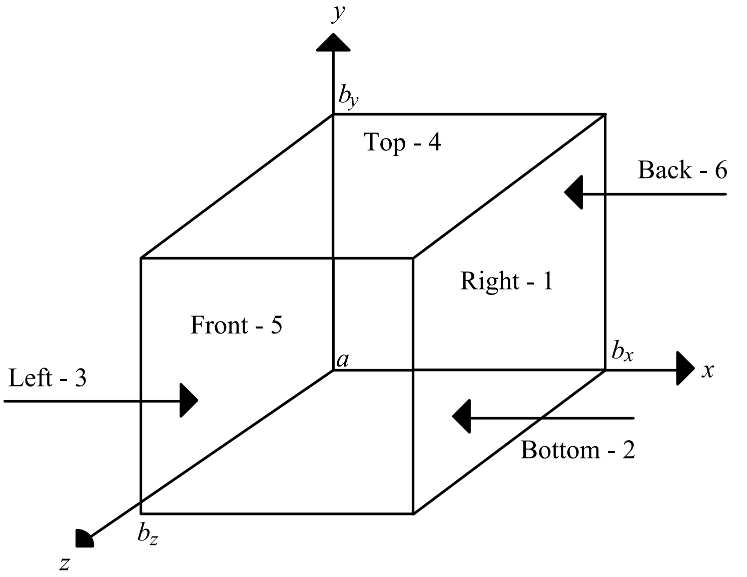
Description

Let c = COEFU, a_x = AX, b_x = BX, n_x = NX, a_y = AY, b_y = BY, n_y = NY, a_z = AZ, b_z = BZ, and n_z = NZ.

FPS3H is based on the code HFFT3D by Boisvert (1984). It solves the equation

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}+cu=p$$

on the domain $(a_x, b_x) \times (a_y, b_y) \times (a_z, b_z)$ (a box) with a user-specified combination of Dirichlet (solution prescribed), Neumann (first derivative prescribed), or periodic boundary conditions. The six sides are numbered as shown in the following diagram.



When $c = 0$ and only Neumann or periodic boundary conditions are prescribed, then any constant may be added to the solution to obtain another solution to the problem. In this case, the solution of minimum ∞-norm is returned.

The solution is computed using either a second-or fourth-order accurate finite-difference approximation of the continuous equation. The resulting system of linear algebraic equations is solved using fast Fourier transform techniques. The algorithm relies upon the fact that $n_x - 1$ and $n_z - 1$ are highly composite (the product of small primes). For details of the algorithm, see Boisvert (1984). If$n_x - 1$ and $n_z - 1$ are highly composite, then the execution time of `FPS3H` is proportional to

$$n_x n_y n_z \left(\log_2^2 n_x + \log_2^2 n_z\right)$$

If evaluations of $p(x, y, z)$ are inexpensive, then the difference in running time between `IORDER` = 2 and `IORDER` = 4 is small.

SLEIG

Determines eigenvalues, eigenfunctions and/or spectral density functions for Sturm-Liouville problems in the form

$$-\frac{d}{dx}(p(x)\frac{du}{dx}) + q(x)u = \lambda r(x)u \text{ for } x \text{ in } (a,b)$$

with boundary conditions (at regular points)

$$a_1 u - a_2 \left(pu' \right) = \lambda \left(a_1' u - a_2' \left(pu' \right) \right) \text{ at } a$$
$$b_1 u + b_2 \left(pu' \right) = 0 \text{ at } b$$

Required Arguments

CONS — Array of size eight containing

$$a_1, a_1', a_2, a_2', b_1, b_2, a \text{ and } b$$

in locations `CONS`(1) through `CONS`(8), respectively. (Input)

COEFFN — User-supplied `SUBROUTINE` to evaluate the coefficient functions. The usage is
`CALL COEFFN (X, PX, QX, RX)`
`X` — Independent variable. (Input)
`PX` — The value of $p(x)$ at `X`. (Output)
`QX` — The value of $q(x)$ at `X`. (Output)
`RX` — The value of $r(x)$ at `X`. (Output)
`COEFFN` must be declared `EXTERNAL` in the calling program.

ENDFIN — Logical array of size two. `ENDFIN(1) = .true.` if the endpoint *a* is finite. `ENDFIN(2) = .true.` if endpoint *b* is finite. (Input)

INDEX — Vector of size `NUMEIG` containing the indices of the desired eigenvalues. (Input)

EVAL — Array of length `NUMEIG` containing the computed approximations to the eigenvalues whose indices are specified in `INDEX`. (Output)

Optional Arguments

NUMEIG — The number of eigenvalues desired. (Input)
Default: `NUMEIG` = size (`INDEX`,1).

TEVLAB — Absolute error tolerance for eigenvalues. (Input)
Default: `TEVLAB` = 10.* machine precision.

TEVLRL — Relative error tolerance for eigenvalues. (Input)
Default: `TEVLRL` = `SQRT`(machine precision).

FORTRAN 90 Interface

Generic: `CALL SLEIG (CONS, COEFFN, ENDFIN, INDEX, EVAL [,…])`

Specific: The specific interface names are `S_SLEIG` and `D_SLEIG`.

FORTRAN 77 Interface

Single: `CALL SLEIG (CONS, COEFFN, ENDFIN, NUMEIG, INDEX, TEVLAB, TEVLRL, EVAL)`

Double: The double precision name is `DSLEIG`.

Example 1

This example computes the first ten eigenvalues of the problem from Titchmarsh (1962) given by

$p(x) = r(x) = 1$

$q(x) = x$

$[a, b] = [0, \infty]$

$u(a) = u(b) = 0$

The eigenvalues are known to be the zeros of

$$f(\lambda) = J_{1/3}\left(\frac{2}{3}\lambda^{3/2}\right) + J_{-1/3}\left(\frac{2}{3}\lambda^{3/2}\right)$$

For each eigenvalue λ_k, the program prints k, λ_k and $f(\lambda_k)$.

```
      USE SLEIG_INT
      USE CBJS_INT
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    I, INDEX(10), NUMEIG
      REAL       CONS(8), EVAL(10), LAMBDA, TEVLAB,&
                 TEVLRL, XNU
      COMPLEX    CBS1(1), CBS2(1), Z
      LOGICAL    ENDFIN(2)
!                                 SPECIFICATIONS FOR INTRINSICS
      INTRINSIC  CMPLX, SQRT
      REAL       SQRT
      COMPLEX    CMPLX
!                                 SPECIFICATIONS FOR SUBROUTINES
!                                 SPECIFICATIONS FOR FUNCTIONS
      EXTERNAL   COEFF
!
      CALL UMACH (2, NOUT)
!                                 Define boundary conditions
      CONS(1) = 1.0
      CONS(2) = 0.0
      CONS(3) = 0.0
      CONS(4) = 0.0
      CONS(5) = 1.0
      CONS(6) = 0.0
      CONS(7) = 0.0
      CONS(8) = 0.0
!
      ENDFIN(1) = .TRUE.
      ENDFIN(2) = .FALSE.
!                                  Compute the first 10 eigenvalues
      NUMEIG = 10
      DO 10  I=1, NUMEIG
         INDEX(I) = I - 1
   10 CONTINUE
!                                  Set absolute and relative tolerance
!
      CALL SLEIG (CONS, COEFF, ENDFIN, INDEX, EVAL)
!
      XNU = -1.0/3.0
      WRITE(NOUT,99998)
      DO 20  I=1, NUMEIG
         LAMBDA = EVAL(I)
         Z      = CMPLX(2.0/3.0*LAMBDA*SQRT(LAMBDA),0.0)
         CALL CBJS (XNU, Z, 1, CBS1)
         CALL CBJS (-XNU, Z, 1, CBS2)
         WRITE (NOUT,99999) I-1, LAMBDA, REAL(CBS1(1) + CBS2(1))
   20 CONTINUE
!
99998 FORMAT(/, 2X, 'index', 5X, 'lambda', 5X, 'f(lambda)',/)
99999 FORMAT(I5, F13.4, E15.4)
      END
!
      SUBROUTINE COEFF (X, PX, QX, RX)
!                                  SPECIFICATIONS FOR ARGUMENTS
      REAL       X, PX, QX, RX
!
      PX = 1.0
      QX = X
      RX = 1.0
      RETURN
      END
```

Output

```
  index     lambda     f(lambda)

    0       2.3381    -0.8285E-05
    1       4.0879    -0.1651E-04
    2       5.5205     0.6843E-04
    3       6.7867    -0.4523E-05
    4       7.9440     0.8952E-04
    5       9.0227     0.1123E-04
    6      10.0401     0.1031E-03
    7      11.0084    -0.7913E-04
    8      11.9361    -0.5095E-04
    9      12.8293     0.4645E-03
```

Comments

1. Workspace may be explicitly provided, if desired, by use of S2EIG/DS2EIG. The reference is:

```
CALL S2EIG (CONS, COEFFN, ENDFIN, NUMEIG, INDEX, TEVLAB, TEVLRL,
EVAL, JOB, IPRINT, TOLS, NUMX, XEF, NRHO, T, TYPE, EF, PDEF,
RHO, IFLAG, WORK, IWORK)
```

The additional arguments are as follows:

JOB — Logical array of length five. (Input)

JOB(1) = .true. if a set of eigenvalues are to be computed but not their eigenfunctions.

JOB(2) = .true. if a set of eigenvalue and eigenfunction pairs are to be computed.

JOB(3) = .true. if the spectral function is to be computed over some subinterval of the essential spectrum.

JOB(4) = .true. if the normal automatic classification is overridden. If JOB(4) = .true. then TYPE(*,*) must be entered correctly. Most users will not want to override the classification process, but it might be appropriate for users experimenting with problems for which the coefficient functions do not have power-like behavior near the singular endpoints. The classification is considered sufficiently important for spectral density function calculations that JOB(4) is ignored with JOB(3) = .true..

JOB(5) = .true. if mesh distribution is chosen by SLEIG. If JOB(5) = .true. and NUMX is zero, the number of mesh points are also chosen by SLEIG. If NUMX > 0 then NUMX mesh points will be used. If JOB(5) = .false., the number NUMX and distribution XEF(*) must be input by the user.

IPRINT — Control levels of internal printing. (Input)

No printing is performed if IPRINT = 0. If either JOB(1) or JOB(2) is true:

IPRINT	**Printed Output**
1	initial mesh (the first 51 or fewer points), eigenvalue estimate at each level
4	the above and at each level matching point for eigenfunction shooting, X(*), EF(*) and PDEF(*) values
5	the above and at each level the brackets for the eigenvalue search, intermediate shooting information for the eigenfunction and eigenfunction norm.

If JOB(3) = .true.

IPRINT	**Printed Output**
1	the actual (a, b) used at each iteration and the total number of eigenvalues computed
2	the above and switchover points to the asymptotic formulas, and some intermediate $\rho(t)$ approximations
4	the above and initial meshes for each iteration, the index of the largest eigenvalue which may be computed, and various eigenvalue and R_N values
4	the above and

$\hat{\rho}$

values at each level

5 the above and R_N add eigenvalues below the switchover point

If JOB(4)=.false.

IPRINT	Printed Output
2	output a description of the spectrum
3	the above and the constants for the Friedrichs' boundary condition(s)
5	the above and intermediate details of the classification calculation

TOLS — Array of length 4 containing tolerances. (Input)

TOLS(1) — absolute error tolerance for eigenfunctions
TOLS(2) — relative error tolerance for eigenfunctions
TOLS(3) — absolute error tolerance for eigenfunction derivatives
TOLS(4) — relative error tolerance for eigenfunction derivatives

The absolute tolerances must be positive.
The relative tolerances must be at least 100 *amach(4)

NUMX — Integer whose value is the number of output points where each eigenfunction is to be evaluated (the number of entries in XEF(*)) when JOB(2) = .true.. If JOB(5)= .false. and NUMX is greater than zero, then NUMX is the number of points in the initial mesh used. If JOB(5) = .false., the points in XEF(*) should be chosen with a reasonable distribution. Since the endpoints *a* and *b* must be part of any mesh, NUMX cannot be one in this case. If JOB(5) = .false. and JOB(3) = .true., then NUMX must be positive. On output, NUMX is set to the number of points for eigenfunctions when input NUMX = 0, and JOB(2) or JOB(5) = .true.. (Input/Output)

XEF — Array of points on input where eigenfunction estimates are desired, if JOB(2) = .true.. Otherwise, if JOB(5) = .false. and NUMX is greater than zero, the user's initial mesh is entered. The entries must be ordered so that a = XEF(1) < XEF(2) < … < XEF(NUMX) = b. If either endpoint is infinite, the corresponding XEF(1) or XEF(NUMX) is ignored. However, it is required that XEF(2) be negative when ENDFIN(1) = .false., and that XEF(NUMX-1) be positive when ENDFIN(2) = .false.. On output, XEF(*) is changed only if JOB(2) and JOB(5) are true. If JOB(2) = .false., this vector is not referenced. If JOB(2) = .true. and NUMX is greater than zero on input, XEF(*) should be dimensioned at least NUMX + 16. If JOB(2) is true and NUMX is zero on input, XEF(*) should be dimensioned at least 31.

NRHO — The number of output values desired for the array RHO(*). NRHO is not used if JOB(3) = .false.. (Input)

T — Real vector of size NRHO containing values where the spectral function RHO(*) is desired. The entries must be sorted in increasing order. The existence and location of a continuous spectrum can be determined by calling SLEIG with the first four entries of JOB set to false and IPRINT set to 1. T(*) is not used if JOB(3) = .false.. (Input)

TYPE — 4 by 2 logical matrix. Column 1 contains information about endpoint *a* and column 2 refers to endpoint *b*.
TYPE(1,*) = .true. if and only if the endpoint is regular
TYPE(2,*) = .true. if and only if the endpoint is limit circle
TYPE(3,*) = .true. if and only if the endpoint is nonoscillatory for all eigenvalues
TYPE(4,*) = .true. if and only if the endpoint is oscillatory for all eigenvalues
Note: all of these values must be correctly input if JOB(4) = .true..
Otherwise, TYPE(*,*) is output. (Input/Output)

EF — Array of eigenfunction values. EF((k − 1)*NUMX + i) is the estimate of u(XEF(i)) corresponding to the eigenvalue in EV(k). If JOB(2) = .false. then this vector is not referenced. If JOB(2) = .true. and NUMX is greater than zero on entry, then EF(*) should be dimensioned at least NUMX * NUMEIG. If JOB(2) = .true. and NUMX is zero on input, then EF(*) should be dimensioned 31 * NUMEIG. (Output)

PDEF — Array of eigenfunction derivative values. PDEF((k-1)*NUMX + i) is the estimate of (*pu*′) (XEF(i)) corresponding to the eigenvalue in EV(k). If JOB(2) = .false. this vector is not referenced. If JOB(2) = .true., it must be dimensioned the same as EF(*). (Output)

RHO — Array of size NRHO containing values for the spectral density function ρ(*t*), RHO(I) = ρ(T(I)). This vector is not referenced if JOB(3) is false. (Output)

IFLAG — Array of size max(1, numeig) containing information about the output. IFLAG(K) refers to the K-th eigenvalue, when JOB(1) or JOB(2) = .true.. Otherwise, only IFLAG(1) is used. Negative values are associated with fatal errors, and the calculations are ceased. Positive values indicate a warning. (Output)
IFLAG(K)

IFLAG(K)	Description
−1	too many levels needed for the eigenvalue calculation; problem seems too difficult at this tolerance. Are the coefficient functions nonsmooth?
−2	too many levels needed for the eigenfunction calculation; problem seems too difficult at this tolerance. Are the eigenfunctions ill-conditioned?
−3	too many levels needed for the spectral density calculation; problem seems too difficult at this tolerance.
−4	the user has requested the spectral density function for a problem which has no continuous spectrum.
−5	the user has requested the spectral density function for a problem with both endpoints generating essential spectrum, i.e. both endpoints either OSC or O-NO.

−6 the user has requested the spectral density function for a problem in spectral category 2 for which a proper normalization of the solution at the NONOSC endpoint is not known; for example, problems with an irregular singular point or infinite endpoint at one end and continuous spectrum generated at the other.

−7 problems were encountered in obtaining a bracket.

−8 too small a step was used in the integration. The `TOLS(*)` values may be too small for this problem.

−9 too small a step was used in the spectral density function calculation for which the continuous spectrum is generated by a finite endpoint.

−10 an argument to the circular trig functions is too large. Try running the problem again with a finer initial mesh or, for singular problems, use interval truncation.

−15 $p(x)$ and $r(x)$ are not positive in the interval (a, b).

−20 eigenvalues and/or eigenfunctions were requested for a problem with an OSC singular endpoint. Interval truncation must be used on such problems.

1 Failure in the bracketing procedure probably due to a cluster of eigenvalues which the code cannot separate. Calculations have continued but any eigenfunction results are suspect. Try running the problem again with tighter input tolerances to separate the cluster.

2 there is uncertainty in the classification for this problem. Because of the limitations of floating point arithmetic, and the nature of the finite sampling, the routine cannot be certain about the classification information at the requested tolerance.

3 there may be some eigenvalues embedded in the essential spectrum. Use of `IPRINT` greater than zero will provide additional output giving the location of the approximating eigenvalues for the step function problem. These could be extrapolated to estimate the actual eigenvalue embedded in the essential spectrum.

4 a change of variables was made to avoid potentially slow convergence. However, the global error estimates may not be as reliable. Some experimentation using different tolerances is recommended.

6 there were problems with eigenfunction convergence in a spectral density calculation. The output $\rho(t)$ may not be accurate.

WORK — Array of size `MAX`(1000, `NUMEIG` + 22) used for workspace.

IWORK — Integer array of size `NUMEIG` + 3 used for workspace.

Description

This subroutine is designed for the calculation of eigenvalues, eigenfunctions and/or spectral density functions for Sturm-Liouville problems in the form

$$-\frac{d}{dx}(p(x)\frac{du}{dx})+q(x)u=\lambda r(x)u \text{ for } x \text{ in } (a,b) \qquad (1)$$

with boundary conditions (at regular points)

$$a_1u-a_2(pu')=\lambda\left(a_1'u-a_2'(pu')\right) \text{ at } a$$

$$b_1u+b_2(pu')=0 \text{ at } b$$

We assume that

$$a_1'a_2-a_1a_2'>0$$

when $a'_1 \neq 0$ and $a'_2 \neq 0$. The problem is considered regular if and only if

- a and b are finite,
- $p(x)$ and $r(x)$ are positive in (a, b),
- $1/p(x)$, $q(x)$ and $r(x)$ are locally integrable near the endpoints.

Otherwise the problem is called singular. The theory assumes that p, p', q, and r are at least continuous on (a, b), though a finite number of jump discontinuities can be handled by suitably defining an input mesh.

For regular problems, there are an infinite number of eigenvalues

$$\lambda_0 < \lambda_1 < \ldots < \lambda_k,\ k \to \infty$$

Each eigenvalue has an associated eigenfunction which is unique up to a constant. For singular problems, there is a wide range in the behavior of the eigenvalues.

As presented in Pruess and Fulton (1993) the approach is to replace (1) by a new problem

$$-\left(\hat{p}\hat{u}'\right)'+\hat{q}\hat{u}=\hat{\lambda}\hat{r}\hat{u} \qquad (2)$$

with analogous boundary conditions

$$a_1\hat{u}(a)-a_2(\hat{p}\hat{u}')(a)=\hat{\lambda}\left[a_1'\hat{u}(a)-a_2{}'(\hat{p}\hat{u}')(a)\right]$$

$$b_1\hat{u}(b)+b_2(\hat{p}\hat{u}')(b)=0$$

where

$$\hat{p}, \hat{q} \text{ and } \hat{r}$$

are step function approximations to *p*, *q*, and *r*, respectively. Given the mesh $a = x_1 < x_2 < \ldots < x_{N+1} = b$, the usual choice for the step functions uses midpoint interpolation, i. e.,

$$\hat{p}(x) = p_n \equiv p(\frac{x_n + x_{n+1}}{2})$$

for *x* in (x_n, x_{n+1}) and similarly for the other coefficient functions. This choice works well for regular problems. Some singular problems require a more sophisticated technique to capture the asymptotic behavior. For the midpoint interpolants, the differential equation (2) has the known closed form solution in

(x_n, x_{n+1})

$$\hat{u}(x) = \hat{u}(x_n)\phi_n'(x - x_n) + (\hat{p}\hat{u}')(x_n)\phi_n(x - x_n) / p_n$$

with

$$\phi_n(t) = \begin{cases} \sin \omega_n t / \omega_n, \tau_n > 0 \\ \sinh \omega_n t / \omega_n, \tau_n < 0 \\ t, \tau = 0 \end{cases}$$

where

$$\tau_n = \left(\hat{\lambda} r_n - q_n\right) / p_n$$

and

$$\omega_n = \sqrt{|\tau_n|}$$

Starting with,

$$\hat{u}(a) \text{ and } (\hat{p}\hat{u}')(a)$$

consistent with the boundary condition,

$$\hat{u}(a) = a_2 - a_2'\hat{\lambda}$$
$$(\hat{p}\hat{u}')(a) = a_1 - a_1'\hat{\lambda}$$

an algorithm is to compute for *n* = 1, 2, ..., *N*,

$$\hat{u}(x_{n+1}) = \hat{u}(x_n)\phi_n'(h_n) + (\hat{p}\hat{u}')(x_n)\phi_n(h_n) / p_n$$
$$(\hat{p}\hat{u}')(x_{n+1}) = -\tau_n p_n \hat{u}(x_n)\phi_n'(h_n) + (\hat{p}\hat{u}')(x_n)\phi_n(h_n)$$

which is a shooting method. For a fixed mesh we can iterate on the approximate eigenvalue until the boundary condition at b is satisfied. This will yield an $O(h^2)$ approximation

$$\hat{\lambda}_k$$

to some λ_k.

The problem (2) has a step spectral function given by

$$\hat{\rho}(t) = \sum \frac{1}{\int \hat{r}(x)\hat{u}_k^2(x)\,dx + \alpha}$$

where the sum is taken over k such that

$$\hat{\lambda}_k \le t$$

and

$$\alpha = a_1' a_2 - a_1 a_2'$$

Additional Examples

Example 2

In this problem from Scott, Shampine and Wing (1969),

$p(x) = r(x) = 1$

$q(x) = x^2 + x^4$

$[a, b] = [-\infty, \infty]$

$u(a) = u(b) = 0$

the first eigenvalue and associated eigenfunction, evaluated at selected points, are computed. As a rough check of the correctness of the results, the magnitude of the residual

$$-\frac{d}{dx}(p(x)\frac{du}{dx}) + q(x)u - \lambda r(x)u$$

is printed. We compute a spline interpolant to u' and use the function CSDER to estimate the quantity $-(p(x)u')'$.

```
      USE S2EIG_INT
      USE CSDER_INT
      USE UMACH_INT
      USE CSAKM_INT
!                                  SPECIFICATIONS FOR LOCAL VARIABLES

      INTEGER    I, IFLAG(1), INDEX(1), IWORK(100), NINTV, NOUT, NRHO, &
                NUMEIG, NUMX
      REAL       BRKUP(61), CONS(8), CSCFUP(4,61), EF(61), EVAL(1), &
                LAMBDA, PDEF(61), PX, QX, RESIDUAL, RHO(1), RX, T(1), &
                TEVLAB, TEVLRL, TOLS(4), WORK(3000), X, XEF(61)
      LOGICAL    ENDFIN(2), JOB(5), TYPE(4,2)
!                                  SPECIFICATIONS FOR INTRINSICS
      INTRINSIC  ABS, REAL
      REAL       ABS, REAL
!                                  SPECIFICATIONS FOR SUBROUTINES
      EXTERNAL   COEFF
!                                  Define boundary conditions
      CONS(1) = 1.0
      CONS(2) = 0.0
      CONS(3) = 0.0
      CONS(4) = 0.0
      CONS(5) = 1.0
      CONS(6) = 0.0
      CONS(7) = 0.0
      CONS(8) = 0.0
!                                  Compute eigenvalue and eigenfunctions
      JOB(1) = .FALSE.
      JOB(2) = .TRUE.
      JOB(3) = .FALSE.
      JOB(4) = .FALSE.
      JOB(5) = .FALSE.
!
      ENDFIN(1) = .FALSE.
      ENDFIN(2) = .FALSE.
!                                  Compute eigenvalue with index 0
      NUMEIG   = 1
      INDEX(1) = 0
!
      TEVLAB  = 1.0E-3
      TEVLRL  = 1.0E-3
      TOLS(1) = TEVLAB
      TOLS(2) = TEVLRL
      TOLS(3) = TEVLAB
      TOLS(4) = TEVLRL
      NRHO    = 0
!                                  Set up mesh, points at which u and
!                                  u' will be computed
      NUMX = 61
      DO 10  I=1, NUMX
         XEF(I) = 0.05*REAL(I-31)
   10 CONTINUE
!
      CALL S2EIG (CONS, COEFF, ENDFIN, NUMEIG, INDEX, TEVLAB, TEVLRL, &
                 EVAL, JOB, 0, TOLS, NUMX, XEF, NRHO, T, TYPE, EF, &
                 PDEF, RHO, IFLAG, WORK, IWORK)
!
      LAMBDA = EVAL(1)
   20 CONTINUE
!                                  Compute spline interpolant to u'
!
      CALL CSAKM (XEF, PDEF, BRKUP, CSCFUP)
      NINTV = NUMX - 1
!
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99997) '     lambda = ', LAMBDA
      WRITE (NOUT,99999)
!                                  At a subset of points from the
!                                  input mesh, compute residual =
!                                  abs( -(u')' + q(x)u - lambda*u ).
!                                  We know p(x) = 1 and r(x) = 1.
      DO 30  I=1, 41, 2
         X = XEF(I+10)
         CALL COEFF (X, PX, QX, RX)
!
!                                  Use the spline fit to u' to
!                                  estimate u'' with CSDER
!
         RESIDUAL = ABS(-CSDER(1,X,BRKUP,CSCFUP)+QX*EF(I+10)- &
                   LAMBDA*EF(I+10))
         WRITE (NOUT,99998) X, EF(I+10), PDEF(I+10), RESIDUAL
   30 CONTINUE
!
99997 FORMAT (/, A14, F10.5, /)
99998 FORMAT (5X, F4.1, 3F15.5)
99999 FORMAT (7X, 'x', 11X, 'u(x)', 10X, 'u''(x)', 9X, 'residual', /)
      END
!
      SUBROUTINE COEFF (X, PX, QX, RX)
!                                  SPECIFICATIONS FOR ARGUMENTS
      REAL       X, PX, QX, RX
!
      PX = 1.0
      QX = X*X + X*X*X*X
      RX = 1.0
      RETURN
      END
```

Output

```
     lambda =    1.39247
         x          u(x)          u'(x)         residual
      -1.0        0.38632        0.65019        0.00189
      -0.9        0.45218        0.66372        0.00081
      -0.8        0.51837        0.65653        0.00023
      -0.7        0.58278        0.62827        0.00113
      -0.6        0.64334        0.57977        0.00183
      -0.5        0.69812        0.51283        0.00230
      -0.4        0.74537        0.42990        0.00273
      -0.3        0.78366        0.33393        0.00265
      -0.2        0.81183        0.22811        0.00273
      -0.1        0.82906        0.11570        0.00278
       0.0        0.83473        0.00000        0.00136
       0.1        0.82893       -0.11568        0.00273
       0.2        0.81170       -0.22807        0.00273
       0.3        0.78353       -0.33388        0.00267
       0.4        0.74525       -0.42983        0.00265
       0.5        0.69800       -0.51274        0.00230
       0.6        0.64324       -0.57967        0.00182
```

```
  0.7         0.58269       -0.62816         0.00113
  0.8         0.51828       -0.65641         0.00023
  0.9         0.45211       -0.66361         0.00081
  1.0         0.38626       -0.65008         0.00189
```

SLCNT

Calculates the indices of eigenvalues of a Sturm-Liouville problem of the form for

$$-\frac{d}{dx}(p(x)\frac{du}{dx})+q(x)u=\lambda r(x)u \text{ for } x \text{ in } [a,b]$$

with boundary conditions (at regular points)

$$a_1u-a_2(pu')=\lambda(a_1'u-a_2'(pu')) \text{ at } a$$

$$b_1u+b_2(pu')=0 \text{ at } b$$

in a specified subinterval of the real line, [α, β].

Required Arguments

ALPHA — Value of the left end point of the search interval. (Input)

BETAR — Value of the right end point of the search interval. (Input)

CONS — Array of size eight containing

$$a_1, a_1', a_2, a_2', b_1, b_2, a \text{ and } b$$

in locations `CONS(1)` … `CONS(8)`, respectively. (Input)

COEFFN — User-supplied `SUBROUTINE` to evaluate the coefficient functions. The usage is

`CALL COEFFN (X, PX, QX, RX)`

`X` – Independent variable. (Input)

`PX` – The value of *p*(*x*) at `X`. (Output)

`QX` – The value of *q*(*x*) at `X`. (Output)

`RX` – The value of *r*(*x*) at `X`. (Output)

`COEFFN` must be declared `EXTERNAL` in the calling program.

ENDFIN — Logical array of size two. `ENDFIN = .true.` if and only if the endpoint *a* is finite. `ENDFIN(2) = .true.` if and only if endpoint *b* is finite. (Input)

IFIRST — The index of the first eigenvalue greater than α. (Output)

NTOTAL — Total number of eigenvalues in the interval [α, β]. (Output)

FORTRAN 90 Interface

Generic: `CALL SLCNT (ALPHA, BETAR, CONS, COEFFN, ENDFIN, IFIRST, NTOTAL)`

Specific: The specific interface names are `S_SLCNT` and `D_SLCNT`.

FORTRAN 77 Interface

Single: `CALL SLCNT (ALPHA, BETAR, CONS, COEFFN, ENDFIN, IFIRST, NTOTAL)`

Double: The double precision name is `DSLCNT`.

Example

Consider the harmonic oscillator (Titchmarsh) defined by

$$p(x) = 1$$

$$q(x) = x^2$$

$$r(x) = 1$$

$$[a, b] = [-\infty, \infty]$$

$$u(a) = 0$$

$$u(b) = 0$$

The eigenvalues of this problem are known to be

$$\lambda_k = 2k + 1, k = 0, 1, \ldots$$

Therefore in the interval [10, 16] we expect `SLCNT` to note three eigenvalues, with the first of these having index five.

```
      USE SLCNT_INT
      USE UMACH_INT
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    IFIRST, NOUT, NTOTAL
      REAL       ALPHA, BETAR, CONS(8)
      LOGICAL    ENDFIN(2)
!                                 SPECIFICATIONS FOR SUBROUTINES
!                                 SPECIFICATIONS FOR FUNCTIONS
      EXTERNAL   COEFFN
!
      CALL UMACH (2, NOUT)
!                                 set u(a) = 0, u(b) = 0
      CONS(1) = 1.0E0
      CONS(2) = 0.0E0
      CONS(3) = 0.0E0
      CONS(4) = 0.0E0
      CONS(5) = 1.0E0
      CONS(6) = 0.0E0
      CONS(7) = 0.0E0
      CONS(8) = 0.0E0
!
      ENDFIN(1) = .FALSE.
      ENDFIN(2) = .FALSE.
!
      ALPHA = 10.0
      BETAR  = 16.0
!
      CALL SLCNT (ALPHA, BETAR, CONS, COEFFN, ENDFIN, IFIRST, NTOTAL)
!
      WRITE (NOUT,99998) ALPHA, BETAR, IFIRST
      WRITE (NOUT,99999) NTOTAL
!
99998 FORMAT (/, 'Index of first eigenvalue in [', F5.2, ',', F5.2, &
            '] IS ', I2)
99999 FORMAT ('Total number of eigenvalues in this interval: ', I2)
!
      END
!
      SUBROUTINE COEFFN (X, PX, QX, RX)
!                                  SPECIFICATIONS FOR ARGUMENTS
      REAL       X, PX, QX, RX
!
      PX = 1.0E0
      QX = X*X
      RX = 1.0E0
      RETURN
      END
```

Output

```
Index of first eigenvalue in [10.00,16.00] is 5
Total number of eigenvalues in this interval: 3
```

Description

This subroutine computes the indices of eigenvalues, if any, in a subinterval of the real line for Sturm-Liouville problems in the form

$$-\frac{d}{dx}(p(x)\frac{du}{dx})+q(x)u=\lambda r(x)u \text{ for } x \text{ in } [a,b]$$

with boundary conditions (at regular points)

$$a_1u-a_2(pu')=\lambda(a_1'u-a_2'(pu')) \text{ at } a$$

$$b_1u+b_2(pu')=0 \text{ at } b$$

It is intended to be used in conjunction with SLEIG, page 973. SLCNT is based on the routine INTERV from the package SLEDGE.

Chapter 6: Transforms

Routines

Usage Notes

Fast Fourier Transforms

A Fast Fourier Transform (FFT) is simply a discrete Fourier transform that can be computed efficiently. Basically, the straightforward method for computing the Fourier transform takes approximately N^2 operations where N is the number of points in the transform, while the FFT (which computes the same values) takes approximately $N \log N$ operations. The algorithms in this chapter are modeled on the Cooley-Tukey (1965) algorithm; hence, the computational savings occur, not for all integers N, but for N which are highly composite. That is, N (or in certain cases $N + 1$ or $N - 1$) should be a product of small primes.

All of the FFT routines compute a *discrete* Fourier transform. The routines accept a vector x of length N and return a vector

$$\hat{x}$$

defined by

$$\hat{x}_m := \sum_{n=1}^{N} x_n \omega_{nm}$$

The various transforms are determined by the selection of ω. In the following table, we indicate the selection of ω for the various transforms. This table should not be mistaken for a definition since the precise transform definitions (at times) depend on whether N or m is even or odd.

Routine	ω_{nm}
FFTRF	cos or sin $\frac{(m-1)(n-1)2\pi}{N}$
FFTRB	cos or sin $\frac{(m-1)(n-1)2\pi}{N}$
FFTCF	$\exp^{-2\pi i(n-1)(m-1)/N}$
FFTCB	$\exp^{2\pi i(n-1)(m-1)/N}$
FSINT	$\sin\frac{nm\pi}{N+1}$
FCOST	$\cos\frac{(n-1)(m-1)\pi}{N-1}$
QSINF	$2\sin\frac{(2m-1)n\pi}{2N}$
QSINB	$4\sin\frac{(2n-1)m\pi}{2N}$
QCOSF	$2\cos\frac{(2m-1)(n-1)\pi}{2N}$
QCOSB	$4\cos\frac{(2\text{n-1})(m-1)\pi}{2N}$

For many of the routines listed above, there is a corresponding “I” (for initialization) routine. Use these routines *only* when repeatedly transforming sequences of the same length. In this situation, the “I” routine will compute the initial setup once, and then the user will call the corresponding “2” routine. This can result in substantial computational savings. For more information on the usage of these routines, the user should consult the documentation under the appropriate routine name.

In addition to the one-dimensional transformations described above, we also provide complex two and three-dimensional FFTs and their inverses based on calls to either FFTCF (page 1017) or FFTCB (page 1019). If you need a higher dimensional transform, then you should consult the example program for FFTCI (page 1022) which suggests a basic strategy one could employ.

Continuous versus Discrete Fourier Transform

There is, of course, a close connection between the discrete Fourier transform and the continuous Fourier transform. Recall that the continuous Fourier transform is defined (Brigham, 1974) as

$$\hat{f}(\omega) = (F\,f)(\omega) = \int_{-\infty}^{\infty} f(t)e^{-2\pi i\omega t}dt$$

We begin by making the following approximation:

$$\begin{aligned}\hat{f}(\omega) &\approx \int_{-T/2}^{T/2} f(t)\, e^{-2\pi i\omega t}\, dt \\ &= \int_0^T f(t-T/2)\, e^{-2\pi i\omega(t-T/2)}\, dt \\ &= e^{\pi i\omega T}\int_0^T f(t-T/2)\, e^{-2\pi i\omega t}\, dt\end{aligned}$$

If we approximate the last integral using the rectangle rule with spacing $h = T/N$, we have

$$\hat{f}(\omega) \approx e^{\pi i\omega T} h \sum_{k=0}^{N-1} e^{-2\pi i\omega kh} f(kh - T/2)$$

Finally, setting $\omega = j/T$ for $j = 0, \ldots, N-1$ yields

$$\hat{f}(j/T) \approx e^{\pi ij} h \sum_{k=0}^{N-1} e^{-2\pi ijk/N} f(kh - T/2) = (-1)^j\, h \sum_{k=0}^{N-1} e^{-2\pi ijk/N} f_k^h$$

where the vector $f^{\,h} = (f(-T/2), \ldots, f((N-1)h - T/2))$. Thus, after scaling the components by $(-1)^j h$, the discrete Fourier transform as computed in `FFTCF` (with input f^h) is related to an approximation of the continuous Fourier transform by the above formula. This is seen more clearly by making a change of variables in the last sum. Set

$$n = k+1,\ m = j+1, \text{ and } f_k^h = x_n$$

then, for $m = 1, \ldots, N$ we have

$$\hat{f}((m-1)/T) \approx -(-1)^m\, h\hat{x}_m = -(-1)^m\, h \sum_{n=1}^{N} e^{-2\pi i(m-1)(n-1)/N} x_n$$

If the function f is expressed as a FORTRAN function routine, then the continuous Fourier transform

$$\hat{f}$$

can be approximated using the IMSL routine `QDAWF` (see Chapter 4, Integration and Differentiation).

Inverse Laplace Transform

The last two routines described in this chapter, `INLAP` (page 1078) and `SINLP` (page 1081), compute the inverse Laplace transforms.

FAST_DFT

Computes the Discrete Fourier Transform (DFT) of a rank-1 complex array, x.

Required Arguments

No required arguments; pairs of optional arguments are required. These pairs are `forward_in` and `forward_out` or `inverse_in` and `inverse_out`.

Optional Arguments

forward_in = x (Input)
Stores the input complex array of rank-1 to be transformed.

forward_out = y (Output)
Stores the output complex array of rank-1 resulting from the transform.

inverse_in = y (Input)
Stores the input complex array of rank-1 to be inverted.

inverse_out = x (Output)
Stores the output complex array of rank-1 resulting from the inverse transform.

ndata = n (Input)
Uses the sub-array of size n for the numbers.
Default value: n = size(x).

ido = ido (Input/Output)
Integer flag that directs user action. Normally, this argument is used only when the working variables required for the transform and its inverse are saved in the calling program unit. Computing the working variables and saving them in internal arrays within fast_dft is the default. This initialization step is expensive.

There is a two-step process to compute the working variables just once. Example 3 illustrates this usage. The general algorithm for this usage is to enter fast_dft with ido = 0. A return occurs thereafter with ido < 0. The optional rank-1 complex array w(:) with size(w) >= –ido must be re-allocated. Then, re-enter fast_dft. The next return from fast_dft has the output value ido = 1. The variables required for the transform and its inverse are saved in w(:). Thereafter, when the routine is entered with ido = 1 and for the same value of n, the contents of w(:) will be used for the working variables. The expensive initialization step is avoided. The optional arguments "ido=" and "work_array=" must be used together.

work_array = w(:) (Output/Input)
Complex array of rank-1 used to store working variables and values between calls to fast_dft. The value for size(w) must be at least as large as the value – ido for the value of ido < 0.

iopt = iopt(:) (Input/Output)
Derived type array with the same precision as the input array; used for passing optional data to fast_dft. The options are as follows:

Packaged Options for FAST_DFT

Option Prefix = ?	Option Name	Option Value
c_, z_	fast_dft_scan_for_NaN	1

Packaged Options for FAST_DFT		
c_, z_	fast_dft_near_power_of_2	2
c_, z_	fast_dft_scale_forward	3
c_, z_	Fast_dft_scale_inverse	4

`iopt(IO) = ?_options(?_fast_dft_scan_for_NaN, ?_dummy)`

Examines each input array entry to find the first value such that

```
isNaN(x(i)) ==.true.
```

```
See the isNaN() function, Chapter 10.
Default: Does not scan for NaNs.
```

`iopt(IO) = ?_options(?_fast_dft_near_power_of_2, ?_dummy)`

Nearest power of 2 ≥ n is returned as an output in `iopt(IO + 1)%idummy`.

`iopt(IO) = ?_options(?_fast_dft_scale_forward, real_part_of_scale)`

`iopt(IO+1) = ?_options(?_dummy, imaginary_part_of_scale)`

Complex number defined by the factor
cmplx(`real_part_of_scale, imaginary_part_of_scale`) is
multiplied by the forward transformed array.
Default value is 1.

`iopt(IO) = ?_options(?_fast_dft_scale_inverse, real_part_of_scale)`

`iopt(IO+1) = ?_options(?_dummy, imaginary_part_of_scale)`

Complex number defined by the factor
cmplx(`real_part_of_scale, imaginary_part_of_scale`) is
multiplied by the inverse transformed array.
Default value is 1.

FORTRAN 90 Interface

Generic: `None`

Specific: The specific interface names are `S_FAST_DFT`, `D_FAST_DFT`, `C_FAST_DFT`, and `Z_FAST_DFT`.

Example 1: Transforming an Array of Random Complex Numbers

An array of random complex numbers is obtained. The transform of the numbers is inverted and the final results are compared with the input array.

```
use fast_dft_int
use rand_gen_int

implicit none
```

```
! This is Example 1 for FAST_DFT.

      integer, parameter :: n=1024
      real(kind(1e0)), parameter :: one=1e0
      real(kind(1e0)) err, y(2*n)
      complex(kind(1e0)), dimension(n) :: a, b, c


! Generate a random complex sequence.
      call rand_gen(y)
      a = cmplx(y(1:n),y(n+1:2*n),kind(one))
      c = a

! Transform and then invert the sequence back.
      call c_fast_dft(forward_in=a, &
           forward_out=b)
      call c_fast_dft(inverse_in=b, &
           inverse_out=a)

! Check that inverse(transform(sequence)) = sequence.
      err = maxval(abs(c-a))/maxval(abs(c))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for FAST_DFT is correct.'
      end if

      end
```

Output

```
Example 1 for FAST_DFT is correct.
```

Description

The `fast_dft` routine is a Fortran 90 version of the FFT suite of IMSL (1994, pp. 772-776). The maximum computing efficiency occurs when the size of the array can be factored in the form

$$n = 2^{i_1}3^{i_2}4^{i_3}5^{i_4}$$

using non-negative integer values $\{i_1, i_2, i_3, i_4\}$. There is no further restriction on $n \geq 1$.

Additional Examples

Example 2: Cyclical Data with a Linear Trend

This set of data is sampled from a function $x(t) = at + b + y(t)$, where $y(t)$ is a harmonic series. The independent variable is normalized as $-1 \leq t \leq 1$. Thus, the data is said *to have cyclical components plus a linear trend*. As a first step, the linear terms are effectively removed from the data using the least-squares system solver `lin_sol_lsq`, Chapter 1. Then, the residuals are transformed and the resulting frequencies are analyzed.

```
      use fast_dft_int
      use lin_sol_lsq_int
      use rand_gen_int
      use sort_real_int

      implicit none

! This is Example 2 for FAST_DFT.

      integer i
      integer, parameter :: n=64, k=4
      integer ip(n)
      real(kind(1e0)), parameter :: one=1e0, two=2e0, zero=0e0
      real(kind(1e0)) delta_t, pi
      real(kind(1e0)) y(k), z(2), indx(k), t(n), temp(n)
      complex(kind(1e0)) a_trend(n,2), a, b_trend(n,1), b, c(k), f(n),&
               r(n), x(n), x_trend(2,1)

! Generate random data for linear trend and harmonic series.
      call rand_gen(z)
      a = z(1); b = z(2)
      call rand_gen(y)
! This emphasizes harmonics 2 through k+1.
      c = y + one

! Determine sampling interval.
      delta_t = two/n
      t=(/(-one+i*delta_t, i=0,n-1)/)

! Compute pi.
      pi = atan(one)*4E0
      indx=(/(i*pi,i=1,k)/)

! Make up data set as a linear trend plus harmonics.
      x = a + b*t + &
         matmul(exp(cmplx(zero,spread(t,2,k)*spread(indx,1,n),kind(one))),c)

! Define least-squares matrix data for a linear trend.
      a_trend(1:,1) = one
      a_trend(1:,2) = t
      b_trend(1:,1) = x

! Solve for a linear trend.
      call lin_sol_lsq(a_trend, b_trend, x_trend)

! Compute harmonic residuals.
      r = x -  reshape(matmul(a_trend,x_trend),(/n/))

! Transform harmonic residuals.
      call c_fast_dft(forward_in=r, forward_out=f)
      ip=(/(i,i=1,n)/)

! The dominant frequencies should be 2 through k+1.
! Sort the magnitude of the transform first.
      call s_sort_real(-(abs(f)), temp, iperm=ip)

! The dominant frequencies are output in ip(1:k).
! Sort these values to compare with 2 through k+1.
      call s_sort_real(real(ip(1:k)), temp)
      ip(1:k)=(/(i,i=2,k+1)/)

! Check the results.
      if (count(int(temp(1:k)) /= ip(1:k)) == 0) then
         write (*,*) 'Example 2 for FAST_DFT is correct.'
      end if

      end
```

Output

```
Example 2 for FAST_DFT is correct.
```

Example 3: Several Transforms with Initialization

In this example, the optional arguments `ido` and `work_array` are used to save working variables in the calling program unit. This results in maximum efficiency of the transform and its inverse since the working variables do not have to be precomputed following each entry to routine `fast_dft`.

```
      use fast_dft_int
      use rand_gen_int

      implicit none

! This is Example 3 for FAST_DFT.

! The value of the array size for work(:) is computed in the
! routine fast_dft as a first step.
      integer, parameter :: n=64
      integer ido_value
      real(kind(1e0)) :: one=1e0
      real(kind(1e0)) err, y(2*n)
      complex(kind(1e0)), dimension(n) :: a, b, save_a
      complex(kind(1e0)), allocatable :: work(:)


! Generate a random complex array.
      call rand_gen(y)
      a = cmplx(y(1:n),y(n+1:2*n),kind(one))
      save_a = a

! Transform and then invert the sequence using the pre-computed
! working values.
      ido_value = 0
      do
         if(allocated(work)) deallocate(work)

! Allocate the space required for work(:).
         if (ido_value <= 0) allocate(work(-ido_value))
         call c_fast_dft(forward_in=a, forward_out=b, &
          ido=ido_value, work_array=work)

         if (ido_value == 1) exit
      end do

! Re-enter routine with working values available in work(:).
      call c_fast_dft(inverse_in=b, inverse_out=a, &
            ido=ido_value, work_array=work)

! Deallocate the space used for work(:).
      if (allocated(work)) deallocate(work)

! Check the results.
      err = maxval(abs(save_a-a))/maxval(abs(save_a))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 3 for FAST_DFT is correct.'
      end if

      end
```

Output

```
Example 3 for FAST_DFT is correct.
```

Example 4: Convolutions using Fourier Transforms

In this example we compute sums

$$c_k = \sum_{j=0}^{n-1} a_j b_{k-j}, k = 0, \ldots, n-1$$

The definition implies a matrix-vector product. A direct approach requires about n^2 operations consisisting of an add and multiply. An efficient method consisting of computing the products of the transforms of the

$$\{a_j\} \text{ and } \{b_j\}$$

then inverting this product, is preferable to the matrix-vector approach for large problems. The example is also illustrated in `operator_ex37`, Chapter 10 using the generic function interface FFT and IFFT.

```
      use fast_dft_int
      use rand_gen_int

      implicit none

! This is Example 4 for FAST_DFT.

      integer j
      integer, parameter :: n=40
      real(kind(1e0)) :: one=1e0
      real(kind(1e0)) err
      real(kind(1e0)), dimension(n) :: x, y, yy(n,n)
      complex(kind(1e0)), dimension(n) :: a, b, c, d, e, f

! Generate two random complex sequence 'a' and 'b'.

      call rand_gen(x)
      call rand_gen(y)
      a=x; b=y

! Compute the convolution 'c' of 'a' and 'b'.
! Use matrix times vector for test results.
      yy(1:,1)=y
      do j=2,n
        yy(2:,j)=yy(1:n-1,j-1)
        yy(1,j)=yy(n,j-1)
      end do

      c=matmul(yy,x)

! Transform the 'a' and 'b' sequences into 'd' and 'e'.

      call c_fast_dft(forward_in=a, &
           forward_out=d)
      call c_fast_dft(forward_in=b, &
           forward_out=e)

! Invert the product d*e.

      call c_fast_dft(inverse_in=d*e, &
           inverse_out=f)

! Check the Convolution Theorem:
! inverse(transform(a)*transform(b)) = convolution(a,b).

      err = maxval(abs(c-f))/maxval(abs(c))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 4 for FAST_DFT is correct.'
      end if

      end
```

Output

```
Example 4 for FAST_DFT is correct.
```

Fatal and Terminal Messages

See the *messages.gls* file for error messages for `fast_dft`. These error messages are numbered 651−661; 701−711.

FAST_2DFT

Computes the Discrete Fourier Transform (2DFT) of a rank-2 complex array, *x*.

Required Arguments

No required arguments; pairs of optional arguments are required. These pairs are `forward_in` and `forward_out` or `inverse_in` and `inverse_out`.

Optional Arguments

`forward_in = x` (Input)
Stores the input complex array of rank-2 to be transformed.

`forward_out = y` (Output)
Stores the output complex array of rank-2 resulting from the transform.

`inverse_in = y` (Input)
Stores the input complex array of rank-2 to be inverted.

`inverse_out = x` (Output)
Stores the output complex array of rank-2 resulting from the inverse transform.

`mdata = m` (Input)
Uses the sub-array in first dimension of size m for the numbers.
Default value: m = size(`x,1`).

`ndata = n` (Input)
Uses the sub-array in the second dimension of size `n` for the numbers.
Default value: `n` = size(`x,2`).

`ido = ido` (Input/Output)
Integer flag that directs user action. Normally, this argument is used only when the working variables required for the transform and its inverse are saved in the calling program unit. Computing the working variables and saving them in internal arrays within `fast_2dft` is the default. This initialization step is expensive.

There is a two-step process to compute the working variables just once. Example 3 illustrates this usage. The general algorithm for this usage is to enter fast_2dft with ido = 0. A return occurs thereafter with ido < 0. The optional rank-1 complex array w(:) with size(w) >= –ido must be re-allocated. Then, re-enter fast_2dft. The next return from fast_2dft has the output value ido = 1. The variables required for the transform and its inverse are saved in w(:). Thereafter, when the routine is entered with ido = 1 and for the same values of m and n, the contents of w(:) will be used for the working variables. The expensive initialization step is avoided. The optional arguments "ido=" and "work_array=" must be used together.

`work_array = w(:)` (Output/Input)

Complex array of rank-1 used to store working variables and values between calls to `fast_2dft`. The value for size(`w`) must be at least as large as the value – `ido` for the value of `ido` < 0.

`iopt = iopt(:)` (Input/Output)

Derived type array with the same precision as the input array; used for passing optional data to `fast_2dft`. The options are as follows:

Packaged Options for `FAST_2DFT`

Option Prefix = ?	Option Name	Option Value
`c_, z_`	`fast_2dft_scan_for_NaN`	1
`c_, z_`	`fast_2dft_near_power_of_2`	2
`c_, z_`	`fast_2dft_scale_forward`	3
`c_, z_`	`fast_2dft_scale_inverse`	4

`iopt(IO) = ?_options(?_fast_2dft_scan_for_NaN, ?_dummy)`

Examines each input array entry to find the first value such that

`isNaN(x(i,j)) ==.true.`

`See the isNaN`() function, Chapter 10.
`Default: Does not scan for NaNs.`

`iopt(IO) = ?_options(?_fast_2dft_near_power_of_2, ?_dummy)`

Nearest powers of 2 ≥ m and ≥ n are returned as an outputs in `iopt(IO + 1)%idummy` and `iopt(IO + 2)%idummy`.

`iopt(IO) = ?_options(?_fast_2dft_scale_forward, real_part_of_scale)`

`iopt(IO+1) = ?_options(?_dummy, imaginary_part_of_scale)`

Complex number defined by the factor
cmplx(`real_part_of_scale, imaginary_part_of_scale`) is
multiplied by the forward transformed array.
Default value is 1.

`iopt(IO) = ?_options(?_fast_2dft_scale_inverse, real_part_of_scale)`

`iopt(IO+1) = ?_options(?_dummy, imaginary_part_of_scale)`

Complex number defined by the factor
cmplx(`real_part_of_scale, imaginary_part_of_scale`) is
multiplied by the inverse transformed array.
Default value is 1.

FORTRAN 90 Interface

Generic: `None`

Specific: The specific interface names are `S_FAST_2DFT`, `D_FAST_2DFT`, `C_FAST_2DFT`, and `Z_FAST_2DFT`.

Example 1: Transforming an Array of Random Complex Numbers

An array of random complex numbers is obtained. The transform of the numbers is inverted and the final results are compared with the input array.

```
      use fast_2dft_int
      use rand_int

      implicit none

! This is Example 1 for FAST_2DFT.

      integer, parameter :: n=24
      integer, parameter :: m=40
      real(kind(1e0)) :: err, one=1e0
      complex(kind(1e0)), dimension(n,m) :: a, b, c


! Generate a random complex sequence.
      a=rand(a); c=a

! Transform and then invert the transform.
      call c_fast_2dft(forward_in=a, &
           forward_out=b)
      call c_fast_2dft(inverse_in=b, &
           inverse_out=a)

! Check that inverse(transform(sequence)) = sequence.
      err = maxval(abs(c-a))/maxval(abs(c))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for FAST_2DFT is correct.'
      end if

     end
```

Output

```
Example 1 for FAST_2DFT is correct.
```

Description

The `fast_2dft` routine is a Fortran 90 version of the FFT suite of IMSL (1994, pp. 772-776).

Additional Examples

Example 2: Cyclical 2D Data with a Linear Trend

This set of data is sampled from a function $x(s, t) = a + bs + ct + y(s, t)$, where $y(s, t)$ is an harmonic series. The independent variables are normalized as

$-1 \le s \le 1$ and $-1 \le t \le 1$. Thus, the data is said *to have cyclical components plus a linear trend*. As a first step, the linear terms are effectively removed from the data using the least-squares system solver . Then, the residuals are transformed and the resulting frequencies are analyzed.

```
      use fast_2dft_int
      use lin_sol_lsq_int
      use sort_real_int
      use rand_int
      implicit none

! This is Example 2 for FAST_2DFT.

      integer i
      integer, parameter :: n=8, k=15
      integer ip(n*n), order(k)
      real(kind(1e0)), parameter :: one=1e0, two=2e0, zero=0e0
      real(kind(1e0)) delta_t
      real(kind(1e0)) rn(3), s(n), t(n), temp(n*n), new_order(k)
      complex(kind(1e0)) a, b, c, a_trend(n*n,3), b_trend(n*n,1),  &
               f(n,n), r(n,n), x(n,n), x_trend(3,1)
      complex(kind(1e0)), dimension(n,n) :: g=zero, h=zero

! Generate random data for planar trend.
      rn = rand(rn)
      a = rn(1)
      b = rn(2)
      c = rn(3)

! Generate the frequency components of the harmonic series.
! Non-zero random amplitudes given on two edges of the square domain.
      g(1:,1)=rand(g(1:,1))
      g(1,1:)=rand(g(1,1:))

! Invert 'g' into the harmonic series 'h' in time domain.
      call c_fast_2dft(inverse_in=g, inverse_out=h)


! Compute sampling interval.
      delta_t = two/n
      s = (/(-one + (i-1)*delta_t, i=1,n)/)
      t = (/(-one + (i-1)*delta_t, i=1,n)/)

! Make up data set as a linear trend plus harmonics.
      x = a + b*spread(s,dim=2,ncopies=n) +   &
             c*spread(t,dim=1,ncopies=n) + h

! Define least-squares matrix data for a planar trend.
      a_trend(1:,1) = one
      a_trend(1:,2) = reshape(spread(s,dim=2,ncopies=n),(/n*n/))
      a_trend(1:,3) = reshape(spread(t,dim=1,ncopies=n),(/n*n/))
      b_trend(1:,1) = reshape(x,(/n*n/))

! Solve for a linear trend.
      call lin_sol_lsq(a_trend, b_trend, x_trend)
```

```
! Compute harmonic residuals.
      r = x -  reshape(matmul(a_trend,x_trend),(/n,n/))

! Transform harmonic residuals.
      call c_fast_2dft(forward_in=r, forward_out=f)

      ip = (/(i,i=1,n**2)/)

! Sort the magnitude of the transform.
      call s_sort_real(-(abs(reshape(f,(/n*n/)))), &
                                  temp, iperm=ip)

! The dominant frequencies are output in ip(1:k).
! Sort these values to compare with the original frequency order.
      call s_sort_real(real(ip(1:k)), new_order)

      order(1:n) = (/(i,i=1,n)/)
      order(n+1:k) = (/((i-n)*n+1,i=n+1,k)/)

! Check the results.
      if (count(order /= int(new_order)) == 0) then
         write (*,*) 'Example 2 for FAST_2DFT is correct.'
      end if

      end
```

Output

```
Example 2 for FAST_2DFT is correct.
```

Example 3: Several 2D Transforms with Initialization

In this example, the optional arguments ido and work_array are used to save working variables in the calling program unit. This results in maximum efficiency of the transform and its inverse since the working variables do not have to be precomputed following each entry to routine fast_2dft.

```
      use fast_2dft_int

      implicit none

! This is Example 3 for FAST_2DFT.

      integer i, j
      integer, parameter :: n=256
      real(kind(1e0)), parameter :: one=1e0, zero=0e0
      real(kind(1e0)) r(n,n), err
      complex(kind(1e0)) a(n,n), b(n,n), c(n,n)

! The value of the array size for work(:) is computed in the
! routine fast_dft as a first step.

      integer ido_value
      complex(kind(1e0)), allocatable :: work(:)
```

```
! Fill in value one for points inside the circle with r=64.
      a = zero
      r = reshape((/(((i-n/2)**2 + (j-n/2)**2, i=1,n), &
                 j=1,n)/),(/n,n/))
      where (r <= (n/4)**2) a = one
      c = a

! Transform and then invert the sequence using the pre-computed
! working values.
      ido_value = 0
      do
         if(allocated(work)) deallocate(work)

! Allocate the space required for work(:).
         if (ido_value <= 0) allocate(work(-ido_value))

! Transform the image and then invert it back.
      call c_fast_2dft(forward_in=a, &
           forward_out=b, IDO=ido_value, work_array=work)
         if (ido_value == 1) exit
      end do
      call c_fast_2dft(inverse_in=b, &
           inverse_out=a, IDO=ido_value, work_array=work)

! Deallocate the space used for work(:).
      if (allocated(work)) deallocate(work)

! Check that inverse(transform(image)) = image.
      err = maxval(abs(c-a))/maxval(abs(c))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 3 for FAST_2DFT is correct.'
      end if

      end
```

Output

```
Example 3 for FAST_2DFT is correct.
```

Fatal and Terminal Messages

See the *messages.gls* file for error messages for `fast_2dft`. These error messages are numbered 670–680; 720–730.

FAST_3DFT

Required Arguments

No required arguments; pairs of optional arguments are required. These pairs are `forward_in` and `forward_out` or `inverse_in` and `inverse_out`.

Optional Arguments

`forward_in = x` (Input)
Stores the input complex array of rank-3 to be transformed.

`forward_out = y` (Output)
Stores the output complex array of rank-3 resulting from the transform.

`inverse_in = y` (Input)
Stores the input complex array of rank-3 to be inverted.

`inverse_out = x` (Output)
Stores the output complex array of rank-3 resulting from the inverse transform.

`mdata = m` (Input)
Uses the sub-array in first dimension of size `m` for the numbers.
Default value: m = size(`x,1`).

`ndata = n` (Input)
Uses the sub-array in the second dimension of size `n` for the numbers.
Default value: `n` = size(`x,2`).

`kdata = k` (Input)
Uses the sub-array in the third dimension of size `k` for the numbers.
Default value: `k` = size(`x,3`).

`ido = ido` (Input/Output)
Integer flag that directs user action. Normally, this argument is used only when the working variables required for the transform and its inverse are saved in the calling program unit. Computing the working variables and saving them in internal arrays within `fast_3dft` is the default. This initialization step is expensive.

There is a two-step process to compute the working variables just once. The general algorithm for this usage is to enter fast_3dft with
ido = 0. A return occurs thereafter with ido < 0. The optional rank-1 complex array w(:) with size(w) >= –ido must be re-allocated. Then, re-enter fast_3dft. The next return from fast_3dft has the output value ido = 1. The variables required for the transform and its inverse are saved in w(:). Thereafter, when the routine is entered with ido = 1 and for the same values of m and n, the contents of w(:) will be used for the working variables. The expensive initialization step

is avoided. The optional arguments "ido=" and "work_array=" must be used together.

`work_array = w(:)` (Output/Input)
Complex array of rank-1 used to store working variables and values between calls to `fast_3dft`. The value for size(`w`) must be at least as large as the value $-$ `ido` for the value of `ido` < 0.

`iopt = iopt(:)` (Input/Output)
Derived type array with the same precision as the input array; used for passing optional data to `fast_3dft`. The options are as follows:

Packaged Options for `FAST_3DFT`		
Option Prefix = ?	Option Name	Option Value
`C_`, `z_`	`fast_3dft_scan_for_NaN`	1
`C_`, `z_`	`fast_3dft_near_power_of_2`	2
`C_`, `z_`	`fast_3dft_scale_forward`	3
`C_`, `z_`	`fast_3dft_scale_inverse`	4

`iopt(IO) = ?_options(?_fast_3dft_scan_for_NaN, ?_dummy)`
Examines each input array entry to find the first value such that

`isNaN(x(i,j,k)) ==.true.`

`See the isNaN() function, Chapter 10.`
`Default: Does not scan for NaNs.`

`iopt(IO) = ?_options(?_fast_3dft_near_power_of_2, ?_dummy)`
Nearest powers of 2 ≥ *m,* ≥ *n,* and ≥ *k* are returned as an outputs in `iopt(IO+1)%idummy`, `iopt(IO+2)%idummy` and `iopt(IO+3)%idummy`

`iopt(IO) = ?_options(?_fast_3dft_scale_forward, real_part_of_scale)`

`iopt(IO+1) = ?_options(?_dummy, imaginary_part_of_scale)`
Complex number defined by the factor
cmplx(`real_part_of_scale, imaginary_part_of_scale`) is
multiplied by the forward transformed array.
Default value is 1.

`iopt(IO) = ?_options(?_fast_3dft_scale_inverse, real_part_of_scale)`

`iopt(IO+1) = ?_options(?_dummy, imaginary_part_of_scale)`
Complex number defined by the factor
cmplx(`real_part_of_scale, imaginary_part_of_scale`) is
multiplied by the inverse transformed array.
Default value is 1.

FORTRAN 90 Interface

Generic: None

Specific: The specific interface names are S_FAST_3DFT, D_FAST_3DFT, C_FAST_3DFT, and Z_FAST_3DFT.

Example 1: Transforming an Array of Random Complex Numbers

An array of random complex numbers is obtained. The transform of the numbers is inverted and the final results are compared with the input array.

```
      use fast_3dft_int

      implicit none

! This is Example 1 for FAST_3DFT.

      integer i, j, k
      integer, parameter :: n=64
      real(kind(1e0)), parameter :: one=1e0, zero=0e0
      real(kind(1e0)) r(n,n,n), err
      complex(kind(1e0)) a(n,n,n), b(n,n,n), c(n,n,n)

! Fill in value one for points inside the sphere
! with radius=16.
      a = zero
      do i=1,n
        do j=1,n
          do k=1,n
            r(i,j,k) = (i-n/2)**2+(j-n/2)**2+(k-n/2)**2
          end do
        end do
      end do
      where (r <= (n/4)**2) a = one
      c = a

! Transform the image and then invert it back.
       call c_fast_3dft(forward_in=a, &
           forward_out=b)
       call c_fast_3dft(inverse_in=b, &
           inverse_out=a)

! Check that inverse(transform(image)) = image.
      err = maxval(abs(c-a))/maxval(abs(c))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for FAST_3DFT is correct.'
      end if

      end
```

Output

Example 1 for FAST_3DFT is correct.

Description

The fast_3dft routine is a Fortran 90 version of the FFT suite of IMSL (1994, pp. 772-776).

Fatal and Terminal Messages

See the *messages.gls* file for error messages for fast_3dft. These error messages are numbered 685–695; 740–750.

FFTRF

Computes the Fourier coefficients of a real periodic sequence.

Required Arguments

N — Length of the sequence to be transformed. (Input)

SEQ — Array of length N containing the periodic sequence. (Input)

COEF — Array of length N containing the Fourier coefficients. (Output)

FORTRAN 90 Interface

Generic: CALL FFTRF (N, SEQ, COEF)

Specific: The specific interface names are S_FFTRF and D_FFTRF.

FORTRAN 77 Interface

Single: CALL FFTRF (N, SEQ, COEF)

Double: The double precision name is DFFTRF.

Example

In this example, a pure cosine wave is used as a data vector, and its Fourier series is recovered. The Fourier series is a vector with all components zero except at the appropriate frequency where it has an *N*.

```
      USE FFTRF_INT
      USE CONST_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=7)
!
      INTEGER    I, NOUT
      REAL       COEF(N), COS, FLOAT, TWOPI, SEQ(N)
      INTRINSIC  COS, FLOAT
      TWOPI = CONST('PI')
!
      TWOPI = 2.0*TWOPI
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 This loop fills out the data vector
!                                 with a pure exponential signal
      DO 10  I=1, N
         SEQ(I) = COS(FLOAT(I-1)*TWOPI/FLOAT(N))
   10 CONTINUE
!                                 Compute the Fourier transform of SEQ
      CALL FFTRF (N, SEQ, COEF)
!                                 Print results
      WRITE (NOUT,99998)
99998 FORMAT (9X, 'INDEX', 5X, 'SEQ', 6X, 'COEF')
      WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
99999 FORMAT (1X, I11, 5X, F5.2, 5X, F5.2)
      END
```

Output

```
INDEX     SEQ      COEF
  1      1.00      0.00
  2      0.62      3.50
  3     -0.22      0.00
  4     -0.90      0.00
  5     -0.90      0.00
  6     -0.22      0.00
  7      0.62      0.00
```

Comments

1. Workspace may be explicitly provided, if desired, by use of F2TRF/DF2TRF. The reference is:

   ```
   CALL F2TRF (N, SEQ, COEF, WFFTR)
   ```

   The additional argument is

   WFFTR — Array of length 2N + 15 initialized by FFTRI (page 1015). (Input)
   The initialization depends on N.

2. The routine FFTRF is most efficient when N is the product of small primes.

3. The arrays COEF and SEQ may be the same.

4. If FFTRF/FFTRB is used repeatedly with the same value of N, then call FFTRI followed by repeated calls to F2TRF/F2TRB. This is more efficient than repeated calls to FFTRF/FFTRB.

Description

The routine FFTRF computes the discrete Fourier transform of a real vector of size *N*. The method used is a variant of the Cooley-Tukey algorithm that is most efficient when *N* is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to *N* log *N*.

Specifically, given an *N*-vector *s* = SEQ, FFTRF returns in *c* = COEF, if *N* is even:

$$c_{2m-2} = \sum_{n=1}^{N} s_n \cos\left[\frac{(m-1)(n-1)2\pi}{N}\right] \qquad m = 2, \ldots, N/2+1$$

$$c_{2m-1} = -\sum_{n=1}^{N} s_n \sin\left[\frac{(m-1)(n-1)2\pi}{N}\right] \qquad m = 2, \ldots, N/2$$

$$c_1 = \sum_{n=1}^{N} s_n$$

If *N* is odd, c_m is defined as above for *m* from 2 to (*N* + 1)/2.

We now describe a fairly common usage of this routine. Let *f* be a real valued function of time. Suppose we sample *f* at *N* equally spaced time intervals of length Δ seconds starting at time t_0. That is, we have

$$\text{SEQ}_{i} := f(t_0 + (i-1)\Delta) \; i = 1, 2, \ldots, N$$

The routine FFTRF treats this sequence as if it were periodic of period *N*. In particular, it assumes that $f(t_0) = f(t_0 + N\Delta)$. Hence, the period of the function is assumed to be $T = N\Delta$.

Now, FFTRF accepts as input SEQ and returns as output coefficients *c* = COEF that satisfy the following relation when *N* is odd (*N* even is similar):

$$\text{SEQ}_i = \frac{1}{N}\left[c_1 + 2\sum_{n=2}^{(N+1)/2} c_{2n-2}\cos\left[\frac{2\pi(n-1)(i-1)}{N}\right] - 2\sum_{n=2}^{(N+1)/2} c_{2n-1}\sin\left[\frac{2\pi(n-1)(i-1)}{N}\right]\right]$$

This formula is very revealing. It can be interpreted in the following manner. The coefficients produced by FFTRF produce an interpolating trigonometric polynomial to the data. That is, if we define

$$\begin{aligned} g(t) \quad &:= \frac{1}{N}\left[c_1 + 2\sum_{n=2}^{(N+1)/2} c_{2n-2}\cos\left[\frac{2\pi(n-1)(t-t_0)}{N\Delta}\right] - 2\sum_{n=2}^{(N+1)/2} c_{2n-1}\sin\left[\frac{2\pi(n-1)(t-t_0)}{N\Delta}\right]\right] \\ &= \frac{1}{N}\left[c_1 + 2\sum_{n=2}^{(N+1)/2} c_{2n-2}\cos\left[\frac{2\pi(n-1)(t-t_0)}{T}\right] - 2\sum_{n=2}^{(N+1)/2} c_{2n-1}\sin\left[\frac{2\pi(n-1)(t-t_0)}{T}\right]\right] \end{aligned}$$

then, we have

$$f(t_0 + (\text{i} - 1)\Delta) = g(t_0 + (i-1)\Delta)$$

Now, suppose we want to discover the dominant frequencies. One forms the vector *P* of length *N*/2 as follows:

$$P_1 := |c_1|$$
$$P_k := \sqrt{c_{2k-2}^2 + c_{2k-1}^2} \qquad k = 2, 3, \ldots, (N+1)/2$$

These numbers correspond to the energy in the spectrum of the signal. In particular, P_k corresponds to the energy level at frequency

$$\frac{k-1}{T} = \frac{k-1}{N\Delta} \qquad k = 1, 2, \ldots, \frac{N+1}{2}$$

Furthermore, note that there are only $(N + 1)/2 \approx T/(2\Delta)$ resolvable frequencies when N observations are taken. This is related to the Nyquist phenomenon, which is induced by discrete sampling of a continuous signal.

Similar relations hold for the case when N is even.

Finally, note that the Fourier transform hsas an (unnormalized) inverse that is implemented in `FFTRB` (page 1012). The routine `FFTRF` is based on the real `FFT` in `FFTPACK`. The package `FFTPACK` was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

FFTRB

Computes the real periodic sequence from its Fourier coefficients.

Required Arguments

N — Length of the sequence to be transformed. (Input)

COEF — Array of length `N` containing the Fourier coefficients. (Input)

SEQ — Array of length `N` containing the periodic sequence. (Output)

FORTRAN 90 Interface

Generic: `CALL FFTRB (N, COEF, SEQ [,…])`

Specific: The specific interface names are `S_FFTRB` and `D_FFTRB`.

FORTRAN 77 Interface

Single: `CALL FFTRB (N, COEF, SEQ)`

Double: The double precision name is `DFFTRB`.

Example

We compute the forward real FFT followed by the inverse operation. In this example, we first compute the Fourier transform

$$\hat{x} = \mathrm{COEF}$$

of the vector x, where $x_j = (-1)^j$ for $j = 1$ to N. This vector

$$\hat{x}$$

is now input into `FFTRB` with the resulting output $s = Nx$, that is, $s_j = (-1)^j N$ for $j = 1$ to N.

```
      USE FFTRB_INT
      USE CONST_INT
      USE FFTRF_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=7)
!
      INTEGER    I, NOUT
      REAL       COEF(N), FLOAT, SEQ(N), TWOPI, X(N)
      INTRINSIC  FLOAT
      TWOPI = CONST('PI')
!
      TWOPI = TWOPI
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Fill the data vector
      DO 10  I=1, N
         X(I) = FLOAT((-1)**I)
   10 CONTINUE
!                                 Compute the forward transform of X
      CALL FFTRF (N, X, COEF)
!                                 Print results
      WRITE (NOUT,99994)
      WRITE (NOUT,99995)
99994 FORMAT (9X, 'Result after forward transform')
99995 FORMAT (9X, 'INDEX', 5X, 'X', 8X, 'COEF')
      WRITE (NOUT,99996) (I, X(I), COEF(I), I=1,N)
99996 FORMAT (1X, I11, 5X, F5.2, 5X, F5.2)
!                                 Compute the backward transform of
!                                 COEF
      CALL FFTRB (N, COEF, SEQ)
!                                Print results
      WRITE (NOUT,99997)
      WRITE (NOUT,99998)
99997 FORMAT (/, 9X, 'Result after backward transform')
99998 FORMAT (9X, 'INDEX', 4X, 'COEF', 6X, 'SEQ')
      WRITE (NOUT,99999) (I, COEF(I), SEQ(I), I=1,N)
99999 FORMAT (1X, I11, 5X, F5.2, 5X, F5.2)
      END
```

Output

```
Result after forward transform
INDEX     X        COEF
  1     -1.00     -1.00
  2      1.00     -1.00
  3     -1.00     -0.48
  4      1.00     -1.00
  5     -1.00     -1.25
  6      1.00     -1.00
  7     -1.00     -4.38

Result after backward transform
INDEX     COEF       SEQ
  1     -1.00     -7.00
  2     -1.00      7.00
  3     -0.48     -7.00
  4     -1.00      7.00
  5     -1.25     -7.00
  6     -1.00      7.00
  7     -4.38     -7.00
```

Comments

1. Workspace may be explicitly provided, if desired, by use of F2TRB/DF2TRB. The reference is:

   ```
   CALL F2TRB (N, COEF, SEQ, WFFTR)
   ```

   The additional argument is

   WFFTR — Array of length 2`N` + 15 initialized by `FFTRI` (page 1015). (Input)
   The initialization depends on `N`.

2. The routine `FFTRB` is most efficient when `N` is the product of small primes.

3. The arrays `COEF` and `SEQ` may be the same.

4. If `FFTRF`/`FFTRB` is used repeatedly with the same value of `N`, then call `FFTRI` (page 1015) followed by repeated calls to `F2TRF`/`F2TRB`. This is more efficient than repeated calls to `FFTRF`/`FFTRB`.

Description

The routine `FFTRB` is the unnormalized inverse of the routine `FFTRF` (page 1009). This routine computes the discrete inverse Fourier transform of a real vector of size *N*. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when *N* is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to *N* log *N*.

Specifically, given an *N*-vector *c* = `COEF`, `FFTRB` returns in *s* = `SEQ`, if *N* is even:

$$s_m = c_1 + (-1)^{(m-1)} c_N + 2\sum_{n=2}^{N/2} c_{2n-2} \cos\frac{\left[(n-1)(m-1)2\pi\right]}{N}$$
$$-2\sum_{n=2}^{N/2} c_{2n-1} \sin\frac{\left[(n-1)(m-1)2\pi\right]}{N}$$

If *N* is odd:

$$
\begin{aligned}
s_m = c_1 + 2 &\sum_{n=2}^{(N+1)/2} c_{2n-2} \cos \frac{\left[(n-1)(m-1)2\pi\right]}{N} \\
-2 &\sum_{n=2}^{(N+1)/2} c_{2n-1} \sin \frac{\left[(n-1)(m-1)2\pi\right]}{N}
\end{aligned}
$$

The routine FFTRB is based on the inverse real FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

FFTRI

Computes parameters needed by FFTRF and FFTRB.

Required Arguments

N — Length of the sequence to be transformed. (Input)

WFFTR — Array of length 2N + 15 containing parameters needed by FFTRF and FFTRB. (Output)

FORTRAN 90 Interface

Generic: CALL FFTRI (N, WFFTR)

Specific: The specific interface names are S_FFTRI and D_FFTRI.

FORTRAN 77 Interface

Single: CALL FFTRI (N, WFFTR)

Double: The double precision name is DFFTRI.

Example

In this example, we compute three distinct real FFTs by calling FFTRI once and then calling F2TRF three times.

```
      USE FFTRI_INT
      USE CONST_INT
      USE F2TRF_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=7)
!
      INTEGER    I, K, NOUT
      REAL       COEF(N), COS, FLOAT, TWOPI, WFFTR(29), SEQ(N)
      INTRINSIC  COS, FLOAT
!
      TWOPI = CONST('PI')
      TWOPI = 2* TWOPI
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Set the work vector
      CALL FFTRI (N, WFFTR)
!
      DO 20  K=1, 3
!                                 This loop fills out the data vector
!                                 with a pure exponential signal
         DO 10  I=1, N
            SEQ(I) = COS(FLOAT(K*(I-1))*TWOPI/FLOAT(N))
   10 CONTINUE
!                                 Compute the Fourier transform of SEQ
         CALL F2TRF (N, SEQ, COEF, WFFTR)
!                                 Print results
         WRITE (NOUT,99998)
99998    FORMAT (/, 9X, 'INDEX', 5X, 'SEQ', 6X, 'COEF')
         WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
99999    FORMAT (1X, I11, 5X, F5.2, 5X, F5.2)
!
   20 CONTINUE
      END
```

Output

```
INDEX     SEQ      COEF
  1      1.00      0.00
  2      0.62      3.50
  3     -0.22      0.00
  4     -0.90      0.00
  5     -0.90      0.00
  6     -0.22      0.00
  7      0.62      0.00



INDEX     SEQ      COEF
  1      1.00      0.00
  2     -0.22      0.00
  3     -0.90      0.00
  4      0.62      3.50
  5      0.62      0.00
  6     -0.90      0.00
  7     -0.22      0.00


INDEX     SEQ      COEF
1      1.00      0.00
2     -0.90      0.00
3      0.62      0.00
4     -0.22      0.00
5     -0.22      0.00
```

```
6       0.62       3.50
7      -0.90       0.00
```

Comments

Different WFFTR arrays are needed for different values of N.

Description

The routine FFTRI initializes the routines FFTRF (page 1009) and FFTRB (page 1012). An efficient way to make multiple calls for the same *N* to routine FFTRF or FFTRB, is to use routine FFTRI for initialization. (In this case, replace FFTRF or FFTRB with F2TRF or F2TRB, respectively.) The routine FFTRI is based on the routine RFFTI in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

FFTCF

Computes the Fourier coefficients of a complex periodic sequence.

Required Arguments

N — Length of the sequence to be transformed. (Input)

SEQ — Complex array of length N containing the periodic sequence. (Input)

COEF — Complex array of length N containing the Fourier coefficients. (Output)

FORTRAN 90 Interface

Generic: CALL FFTCF (N, SEQ, COEF)

Specific: The specific interface names are S_FFTCF and D_FFTCF.

FORTRAN 77 Interface

Single: CALL FFTCF (N, SEQ, COEF)

Double: The double precision name is DFFTCF.

Example

In this example, we input a pure exponential data vector and recover its Fourier series, which is a vector with all components zero except at the appropriate frequency where it has an *N*. Notice that the norm of the input vector is

$$\sqrt{N}$$

but the norm of the output vector is *N*.

```
      USE FFTCF_INT
      USE CONST_INT
      USE UMACH_INT

      INTEGER    N
      PARAMETER  (N=7)
!
      INTEGER    I, NOUT
      REAL       TWOPI
      COMPLEX    C, CEXP, COEF(N), H, SEQ(N)
      INTRINSIC  CEXP
!
      C     = (0.,1.)
      TWOPI = CONST('PI')
      TWOPI = 2.0 * TWOPI
!                                 Here we compute (2*pi*i/N)*3.
      H = (TWOPI*C/N)*3.
!                                 This loop fills out the data vector
!                                 with a pure exponential signal of
!                                 frequency 3.
      DO 10  I=1, N
         SEQ(I) = CEXP((I-1)*H)
   10 CONTINUE
!                                 Compute the Fourier transform of SEQ
      CALL FFTCF (N, SEQ, COEF)
!                                 Get output unit number and print
!                                 results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99998)
99998 FORMAT (9X, 'INDEX', 8X, 'SEQ', 15X, 'COEF')
      WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
99999 FORMAT (1X, I11, 5X,'(',F5.2,',',F5.2,')', &
                     5X,'(',F5.2,',',F5.2,')')
      END
```

Output

```
INDEX        SEQ               COEF
  1     ( 1.00, 0.00)     ( 0.00, 0.00)
  2     (-0.90, 0.43)     ( 0.00, 0.00)
  3     ( 0.62,-0.78)     ( 0.00, 0.00)
  4     (-0.22, 0.97)     ( 7.00, 0.00)
  5     (-0.22,-0.97)     ( 0.00, 0.00)
  6     ( 0.62, 0.78)     ( 0.00, 0.00)
  7     (-0.90,-0.43)     ( 0.00, 0.00)
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `F2TCF/DF2TCF`. The reference is:

   ```
   CALL F2TCF (N, SEQ, COEF, WFFTC, CPY)
   ```

   The additional arguments are as follows:

WFFTC — Real array of length 4 * `N` + 15 initialized by `FFTCI` (page 1022). The initialization depends on `N`. (Input)

CPY — Real array of length 2 * `N`. (Workspace)

2. The routine `FFTCF` is most efficient when `N` is the product of small primes.

3. The arrays `COEF` and `SEQ` may be the same.

4. If `FFTCF`/`FFTCB` is used repeatedly with the same value of `N`, then call `FFTCI` followed by repeated calls to `F2TCF`/`F2TCB`. This is more efficient than repeated calls to `FFTCF`/`FFTCB`.

Description

The routine `FFTCF` computes the discrete complex Fourier transform of a complex vector of size *N*. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when *N* is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to *N* log *N*. This considerable savings has historically led people to refer to this algorithm as the "fast Fourier transform" or FFT.

Specifically, given an *N*-vector *x*, `FFTCF` returns in *c* = `COEF`

$$c_m = \sum_{n=1}^{N} x_n e^{-2\pi i(n-1)(m-1)/N}$$

Furthermore, a vector of Euclidean norm *S* is mapped into a vector of norm

$$\sqrt{N}S$$

Finally, note that we can invert the Fourier transform as follows:

$$x_n = \frac{1}{N}\sum_{m=1}^{N} c_m e^{2\pi i(m-1)(n-1)/N}$$

This formula reveals the fact that, after properly normalizing the Fourier coefficients, one has the coefficients for a trigonometric interpolating polynomial to the data. An unnormalized inverse is implemented in `FFTCB` (page 1019). `FFTCF` is based on the complex FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

FFTCB

Computes the complex periodic sequence from its Fourier coefficients.

Required Arguments

N — Length of the sequence to be transformed. (Input)

COEF — Complex array of length `N` containing the Fourier coefficients. (Input)

SEQ — Complex array of length `N` containing the periodic sequence. (Output)

FORTRAN 90 Interface

Generic: `CALL FFTCB (N, COEF, SEQ)`

Specific: The specific interface names are `S_FFTCB` and `D_FFTCB`.

FORTRAN 77 Interface

Single: `CALL FFTCB (N, COEF, SEQ)`

Double: The double precision name is `DFFTCB`.

Example

In this example, we first compute the Fourier transform of the vector *x*, where $x_j = j$ for $j = 1$ to *N*. Note that the norm of *x* is $(N[N + 1][2N + 1]/6)^{1/2}$, and hence, the norm of the transformed vector

$$\hat{x} = c$$

is $N([N + 1][2N + 1]/6)^{1/2}$. The vector

$$\hat{x}$$

is used as input into `FFTCB` with the resulting output $s = Nx$, that is, $s_j = jN$, for $j = 1$ to *N*.

```
      USE FFTCB_INT
      USE FFTCF_INT
      USE UMACH_INT

      INTEGER    N
      PARAMETER  (N=7)
!
      INTEGER    I, NOUT
      COMPLEX    CMPLX, SEQ(N), COEF(N), X(N)
      INTRINSIC  CMPLX
!                                 This loop fills out the data vector
!                                 with X(I)=I, I=1,N
      DO 10  I=1, N
         X(I) = CMPLX(I,0)
   10 CONTINUE
!                                 Compute the forward transform of X
      CALL FFTCF (N, X, COEF)
!                                 Compute the backward transform of
!                                 COEF
      CALL FFTCB (N, COEF, SEQ)
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Print results
      WRITE (NOUT,99998)
      WRITE (NOUT,99999) (I, X(I), COEF(I), SEQ(I), I=1,N)
```

```
99998 FORMAT (5X, 'INDEX', 9X, 'INPUT', 9X, 'FORWARD TRANSFORM', 3X, &
             'BACKWARD TRANSFORM')
99999 FORMAT (1X, I7, 7X,'(',F5.2,',',F5.2,')', &
                    7X,'(',F5.2,',',F5.2,')', &
                    7X,'(',F5.2,',',F5.2,')')
      END
```

Output

```
INDEX      INPUT         FORWARD TRANSFORM   BACKWARD TRANSFORM
 1       ( 1.00, 0.00)       (28.00, 0.00)        ( 7.00, 0.00)
 2       ( 2.00, 0.00)       (-3.50, 7.27)        (14.00, 0.00)
 3       ( 3.00, 0.00)       (-3.50, 2.79)        (21.00, 0.00)
 4       ( 4.00, 0.00)       (-3.50, 0.80)        (28.00, 0.00)
 5       ( 5.00, 0.00)       (-3.50,-0.80)        (35.00, 0.00)
 6       ( 6.00, 0.00)       (-3.50,-2.79)        (42.00, 0.00)
 7       ( 7.00, 0.00)       (-3.50,-7.27)        (49.00, 0.00)
```

Comments

1. Workspace may be explicitly provided, if desired, by use of F2TCB/DF2TCB. The reference is:

   CALL F2TCB (N, COEF, SEQ, WFFTC, CPY)

   The additional arguments are as follows:

   WFFTC — Real array of length 4 * N + 15 initialized by FFTCI (page 1022). The initialization depends on N. (Input)

   CPY — Real array of length 2 * N. (Workspace)

2. The routine FFTCB is most efficient when N is the product of small primes.

3. The arrays COEF and SEQ may be the same.

4. If FFTCF/FFTCB is used repeatedly with the same value of N; then call FFTCI followed by repeated calls to F2TCF/F2TCB. This is more efficient than repeated calls to FFTCF/FFTCB.

Description

The routine FFTCB computes the inverse discrete complex Fourier transform of a complex vector of size *N*. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when *N* is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to *N* log *N*. This considerable savings has historically led people to refer to this algorithm as the "fast Fourier transform" or FFT.

Specifically, given an *N*-vector *c* = COEF, FFTCB returns in *s* = SEQ

$$s_m = \sum_{n=1}^{N} c_n e^{2\pi i(n-1)(m-1)/N}$$

Furthermore, a vector of Euclidean norm S is mapped into a vector of norm

$$\sqrt{N}S$$

Finally, note that we can invert the inverse Fourier transform as follows:

$$c_n = \frac{1}{N}\sum_{m=1}^{N} s_m e^{-2\pi i(n-1)(m-1)/N}$$

This formula reveals the fact that, after properly normalizing the Fourier coefficients, one has the coefficients for a trigonometric interpolating polynomial to the data. `FFTCB` is based on the complex inverse FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

FFTCI

Computes parameters needed by `FFTCF` and `FFTCB`.

Required Arguments

N — Length of the sequence to be transformed. (Input)

WFFTC — Array of length 4`N` + 15 containing parameters needed by `FFTCF` and `FFTCB`. (Output)

FORTRAN 90 Interface

Generic: `CALL FFTCI (N, WFFTC)`

Specific: The specific interface names are `S_FFTCI` and `D_FFTCI`.

FORTRAN 77 Interface

Single: `CALL FFTCI (N, WFFTC)`

Double: The double precision name is `DFFTCI`.

Example

In this example, we compute a two-dimensional complex FFT by making one call to `FFTCI` followed by 2`N` calls to `F2TCF`.

```
      USE FFTCI_INT
      USE CONST_INT
      USE F2TCF_INT
      USE UMACH_INT
!                                 SPECIFICATIONS FOR PARAMETERS
      INTEGER    N
      PARAMETER  (N=4)
!
```

```
      INTEGER    I, IR, IS, J, NOUT
      REAL       FLOAT, TWOPI, WFFTC(35), CPY(2*N)
      COMPLEX    CEXP, CMPLX, COEF(N,N), H, SEQ(N,N), TEMP
      INTRINSIC  CEXP, CMPLX, FLOAT
!
      TWOPI = CONST(’PI’)
      TWOPI = 2*TWOPI
      IR    = 3
      IS    = 1
!                                 Here we compute e**(2*pi*i/N)
      TEMP = CMPLX(0.0,TWOPI/FLOAT(N))
      H    = CEXP(TEMP)
!                                 Fill SEQ with data
      DO 20  I=1, N
         DO 10  J=1, N
            SEQ(I,J) = H**((I-1)*(IR-1)+(J-1)*(IS-1))
   10 CONTINUE
   20 CONTINUE
!                                 Print out SEQ
!                                 Get output unit number
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99997)
      DO 30  I=1, N
         WRITE (NOUT,99998) (SEQ(I,J),J=1,N)
   30 CONTINUE
!                                 Set initialization vector
      CALL FFTCI (N, WFFTC)
!                                 Transform the columns of SEQ
      DO 40  I=1, N
         CALL F2TCF (N, SEQ(1:,I), COEF(1:,I), WFFTC, CPY)
   40 CONTINUE
!                                 Take transpose of the result
      DO 60  I=1, N
         DO 50  J=I + 1, N
            TEMP      = COEF(I,J)
            COEF(I,J) = COEF(J,I)
            COEF(J,I) = TEMP
   50  CONTINUE
   60 CONTINUE
!                                 Transform the columns of this result
      DO 70  I=1, N
         CALL F2TCF (N, COEF(1:,I), SEQ(1:,I), WFFTC, CPY)
   70 CONTINUE
!                                 Take transpose of the result
      DO 90  I=1, N
         DO 80  J=I + 1, N
            TEMP     = SEQ(I,J)
            SEQ(I,J) = SEQ(J,I)
            SEQ(J,I) = TEMP
   80  CONTINUE
   90 CONTINUE
!                                 Print results
      WRITE (NOUT,99999)
      DO 100  I=1, N
         WRITE (NOUT,99998) (SEQ(I,J),J=1,N)
```

```
  100 CONTINUE
!
99997 FORMAT (1X, 'The input matrix is below')
99998 FORMAT (1X, 4(' (',F5.2,',',F5.2,')'))
99999 FORMAT (/, 1X, 'Result of two-dimensional transform')
      END
```

Output

```
The input matrix is below
 ( 1.00, 0.00) ( 1.00, 0.00) ( 1.00, 0.00) ( 1.00, 0.00)
 (-1.00, 0.00) (-1.00, 0.00) (-1.00, 0.00) (-1.00, 0.00)
 ( 1.00, 0.00) ( 1.00, 0.00) ( 1.00, 0.00) ( 1.00, 0.00)
 (-1.00, 0.00) (-1.00, 0.00) (-1.00, 0.00) (-1.00, 0.00)

Result of two-dimensional transform
 ( 0.00, 0.00) ( 0.00, 0.00) ( 0.00, 0.00) ( 0.00, 0.00)
 ( 0.00, 0.00) ( 0.00, 0.00) ( 0.00, 0.00) ( 0.00, 0.00)
 (16.00, 0.00) ( 0.00, 0.00) ( 0.00, 0.00) ( 0.00, 0.00)
 ( 0.00, 0.00) ( 0.00, 0.00) ( 0.00, 0.00) ( 0.00, 0.00)
```

Comments

Different `WFFTC` arrays are needed for different values of `N`.

Description

The routine `FFTCI` initializes the routines `FFTCF` (page 1017) and `FFTCB` (page 1019). An efficient way to make multiple calls for the same `N` to IMSL routine `FFTCF` or `FFTCB` is to use routine `FFTCI` for initialization. (In this case, replace `FFTCF` or `FFTCB` with `F2TCF` or `F2TCB`, respectively.) The routine `FFTCI` is based on the routine `CFFTI` in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

FSINT

Computes the discrete Fourier sine transformation of an odd sequence.

Required Arguments

N — Length of the sequence to be transformed. It must be greater than 1. (Input)

SEQ — Array of length `N` containing the sequence to be transformed. (Input)

COEF — Array of length `N` + 1 containing the transformed sequence. (Output)

FORTRAN 90 Interface

Generic: `CALL FSINT (N, SEQ, COEF)`

Specific: The specific interface names are `S_FSINT` and `D_FSINT`.

FORTRAN 77 Interface

Single: CALL FSINT (N, SEQ, COEF)

Double: The double precision name is DFSINT.

Example

In this example, we input a pure sine wave as a data vector and recover its Fourier sine series, which is a vector with all components zero except at the appropriate frequency it has an *N*.

```
      USE FSINT_INT
      USE CONST_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=7)
!
      INTEGER    I, NOUT
      REAL       COEF(N+1), FLOAT, PI, SIN, SEQ(N)
      INTRINSIC  FLOAT, SIN
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Fill the data vector SEQ
!                                 with a pure sine wave
      PI = CONST('PI')
      DO 10  I=1, N
         SEQ(I) = SIN(FLOAT(I)*PI/FLOAT(N+1))
   10 CONTINUE
!                                 Compute the transform of SEQ
      CALL FSINT (N, SEQ, COEF)
!                                 Print results
      WRITE (NOUT,99998)
      WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
99998 FORMAT (9X, 'INDEX', 6X, 'SEQ', 7X, 'COEF')
99999 FORMAT (1X, I11, 5X, F6.2, 5X, F6.2)
      END
```

Output

```
INDEX      SEQ       COEF
  1       0.38       8.00
  2       0.71       0.00
  3       0.92       0.00
  4       1.00       0.00
  5       0.92       0.00
  6       0.71       0.00
  7       0.38       0.00
```

Comments

1. Workspace may be explicitly provided, if desired, by use of F2INT/DF2INT. The reference is:

   ```
   CALL F2INT (N, SEQ, COEF, WFSIN)
   ```

The additional argument is:

WFSIN — Array of length `INT(2.5 * N + 15)` initialized by `FSINI`. The initialization depends on `N`. (Input)

2. The routine `FSINT` is most efficient when `N` + 1 is the product of small primes.

3. The routine `FSINT` is its own (unnormalized) inverse. Applying `FSINT` twice will reproduce the original sequence multiplied by 2 * (`N` + 1).

4. The arrays `COEF` and `SEQ` may be the same, if `SEQ` is also dimensioned at least `N` + 1.

5. `COEF` (`N` + 1) is needed as workspace.

6. If `FSINT` is used repeatedly with the same value of `N`, then call `FSINI` (page 1026) followed by repeated calls to `F2INT`. This is more efficient than repeated calls to `FSINT`.

Description

The routine `FSINT` computes the discrete Fourier sine transform of a real vector of size *N*. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when *N* + 1 is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to *N* log *N*.

Specifically, given an *N*-vector *s* = `SEQ`, `FSINT` returns in *c* = `COEF`

$$c_m = 2\sum_{n=1}^{N} s_n \sin\left(\frac{mn\pi}{N+1}\right)$$

Finally, note that the Fourier sine transform is its own (unnormalized) inverse. The routine `FSINT` is based on the sine FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

FSINI

Computes parameters needed by `FSINT`.

Required Arguments

N — Length of the sequence to be transformed. `N` must be greater than 1. (Input)

WFSIN — Array of length `INT`(2.5 * `N` + 15) containing parameters needed by `FSINT`. (Output)

FORTRAN 90 Interface

Generic: `CALL FSINI (N, WFSIN)`

Specific: The specific interface names are S_FSINI and D_FSINI.

FORTRAN 77 Interface

Single: CALL FSINI (N, WFSIN)

Double: The double precision name is DFSINI.

Example

In this example, we compute three distinct sine FFTs by calling FSINI once and then calling F2INT three times.

```
      USE FSINI_INT
      USE UMACH_INT
      USE CONST_INT
      USE F2INT_INT
      INTEGER    N
      PARAMETER  (N=7)
!
      INTEGER    I, K, NOUT
      REAL       COEF(N+1), FLOAT, PI, SIN, WFSIN(32), SEQ(N)
      INTRINSIC  FLOAT, SIN
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Initialize the work vector WFSIN
      CALL FSINI (N, WFSIN)
!                                 Different frequencies of the same
!                                 wave will be transformed
      DO 20  K=1, 3
!                                 Fill the data vector SEQ
!                                 with a pure sine wave
         PI = CONST('PI')
         DO 10  I=1, N
            SEQ(I) = SIN(FLOAT(K*I)*PI/FLOAT(N+1))
   10    CONTINUE
!                                 Compute the transform of SEQ
         CALL F2INT (N, SEQ, COEF, WFSIN)
!                                 Print results
         WRITE (NOUT,99998)
         WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
   20 CONTINUE
99998 FORMAT (/, 9X, 'INDEX', 6X, 'SEQ', 7X, 'COEF')
99999 FORMAT (1X, I11, 5X, F6.2, 5X, F6.2)
      END
```

Output

```
INDEX      SEQ       COEF
  1       0.38       8.00
  2       0.71       0.00
  3       0.92       0.00
  4       1.00       0.00
  5       0.92       0.00
  6       0.71        0.00
  7       0.38        0.00

INDEX       SEQ        COEF
  1       0.71        0.00
  2       1.00        8.00
  3       0.71        0.00
  4       0.00        0.00
  5      -0.71        0.00
  6      -1.00        0.00
  7      -0.71        0.00

INDEX       SEQ        COEF
  1       0.92        0.00
  2       0.71        0.00
  3      -0.38        8.00
  4      -1.00        0.00
  5      -0.38        0.00
  6       0.71        0.00
  7       0.92        0.00
```

Comments

Different `WFSIN` arrays are needed for different values of `N`.

Description

The routine `FSINI` initializes the routine `FSINT` (page 1024). An efficient way to make multiple calls for the same *N* to IMSL routine `FSINT`, is to use routine `FSINI` for initialization. (In this case, replace `FSINT` with `F2INT`.) The routine `FSINI` is based on the routine `SINTI` in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

FCOST

Computes the discrete Fourier cosine transformation of an even sequence.

Required Arguments

N — Length of the sequence to be transformed. It must be greater than 1.   (Input)

SEQ — Array of length `N` containing the sequence to be transformed.   (Input)

COEF — Array of length `N` containing the transformed sequence.   (Output)

FORTRAN 90 Interface

Generic: `CALL FCOST (N, SEQ, COEF)`

Specific: The specific interface names are `S_FCOST` and `D_FCOST`.

FORTRAN 77 Interface

Single: `CALL FCOST (N, SEQ, COEF)`

Double: The double precision name is `DFCOST`.

Example

In this example, we input a pure cosine wave as a data vector and recover its Fourier cosine series, which is a vector with all components zero except at the appropriate frequency it has an $N - 1$.

```
      USE FCOST_INT
      USE CONST_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=7)
!
      INTEGER    I, NOUT
      REAL       COEF(N), COS, FLOAT, PI, SEQ(N)
      INTRINSIC  COS, FLOAT
!
      CALL UMACH (2, NOUT)
!                                 Fill the data vector SEQ
!                                 with a pure cosine wave
      PI = CONST('PI')
      DO 10  I=1, N
         SEQ(I) = COS(FLOAT(I-1)*PI/FLOAT(N-1))
   10 CONTINUE
!                                 Compute the transform of SEQ
      CALL FCOST (N, SEQ, COEF)
!                                 Print results
      WRITE (NOUT,99998)
      WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
99998 FORMAT (9X, 'INDEX', 6X, 'SEQ', 7X, 'COEF')
99999 FORMAT (1X, I11, 5X, F6.2, 5X, F6.2)
      END
```

Output

```
INDEX      SEQ       COEF
  1       1.00       0.00
  2       0.87       6.00
  3       0.50       0.00
  4       0.00       0.00
  5      -0.50       0.00
  6      -0.87       0.00
  7      -1.00       0.00
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `F2OST/DF2OST`. The reference is:

CALL F2OST (N, SEQ, COEF, WFCOS)

The additional argument is

WFCOS — Array of length 3 * N + 15 initialized by FCOSI (page 1030). The initialization depends on N. (Input)

2. The routine FCOST is most efficient when N – 1 is the product of small primes.

3. The routine FCOST is its own (unnormalized) inverse. Applying FCOST twice will reproduce the original sequence multiplied by 2 * (N – 1).

4. The arrays COEF and SEQ may be the same.

5. If FCOST is used repeatedly with the same value of N, then call FCOSI followed by repeated calls to F2OST. This is more efficient than repeated calls to FCOST.

Description

The routine FCOST computes the discrete Fourier cosine transform of a real vector of size *N*. The method used is a variant of the Cooley-Tukey algorithm , which is most efficient when *N* – 1 is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to *N* log *N*.

Specifically, given an *N*-vector *s* = SEQ, FCOST returns in *c* = COEF

$$c_m = 2\sum_{n=2}^{N-1} s_n \cos\left[\frac{(m-1)(n-1)\pi}{N-1}\right] + s_1 + s_N(-1)^{(m-1)}$$

Finally, note that the Fourier cosine transform is its own (unnormalized) inverse. Two applications of FCOST to a vector *s* produces (2*N* – 2)*s*. The routine FCOST is based on the cosine FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

FCOSI

Computes parameters needed by FCOST.

Required Arguments

N — Length of the sequence to be transformed. N must be greater than 1. (Input)

WFCOS — Array of length 3N + 15 containing parameters needed by FCOST. (Output)

FORTRAN 90 Interface

Generic: CALL FCOSI (N, WFCOS)

Specific: The specific interface names are S_FCOSI and D_FCOSI.

FORTRAN 77 Interface

Single: CALL FCOSI (N, WFCOS)

Double: The double precision name is DFCOSI.

Example

In this example, we compute three distinct cosine FFTs by calling FCOSI once and then calling F2OST three times.

```
      USE FCOSI_INT
      USE CONST_INT
      USE F2OST_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=7)
!
      INTEGER    I, K, NOUT
      REAL       COEF(N), COS, FLOAT, PI, WFCOS(36), SEQ(N)
      INTRINSIC  COS, FLOAT
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Initialize the work vector WFCOS
      CALL FCOSI (N, WFCOS)
!                                 Different frequencies of the same
!                                 wave will be transformed
      PI = CONST(’PI’)
      DO 20  K=1, 3
!                                 Fill the data vector SEQ
!                                 with a pure cosine wave
         DO 10  I=1, N
            SEQ(I) = COS(FLOAT(K*(I-1))*PI/FLOAT(N-1))
   10    CONTINUE
!                                 Compute the transform of SEQ
         CALL F2OST (N, SEQ, COEF, WFCOS)
!                                 Print results
         WRITE (NOUT,99998)
         WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
   20 CONTINUE
99998 FORMAT (/, 9X, ’INDEX’, 6X, ’SEQ’, 7X, ’COEF’)
99999 FORMAT (1X, I11, 5X, F6.2, 5X, F6.2)
      END
```

Output

```
INDEX      SEQ       COEF
  1       1.00       0.00
  2       0.87       6.00
  3       0.50       0.00
  4       0.00       0.00
  5      -0.50       0.00
  6      -0.87       0.00
  7      -1.00       0.00

INDEX      SEQ       COEF
  1       1.00       0.00
  2       0.50       0.00
  3      -0.50       6.00
  4      -1.00       0.00
  5      -0.50       0.00
  6       0.50       0.00
  7       1.00       0.00

INDEX      SEQ       COEF
  1       1.00       0.00
  2       0.00       0.00
  3      -1.00       0.00
  4       0.00       6.00
  5       1.00       0.00
  6       0.00       0.00
  7      -1.00       0.00
```

Comments

Different `WFCOS` arrays are needed for different values of `N`.

Description

The routine `FCOSI` initializes the routine `FCOST` (page 1028). An efficient way to make multiple calls for the same *N* to IMSL routine `FCOST` is to use routine `FCOSI` for initialization. (In this case, replace `FCOST` with `F2OST`.) The routine `FCOSI` is based on the routine `COSTI` in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

QSINF

Computes the coefficients of the sine Fourier transform with only odd wave numbers.

Required Arguments

N — Length of the sequence to be transformed. (Input)

SEQ — Array of length `N` containing the sequence. (Input)

COEF — Array of length `N` containing the Fourier coefficients. (Output)

FORTRAN 90 Interface

Generic: `CALL QSINF (N, SEQ, COEF)`

Specific: The specific interface names are `S_QSINF` and `D_QSINF`.

FORTRAN 77 Interface

Single: CALL QSINF (N, SEQ, COEF)

Double: The double precision name is DQSINF.

Example

In this example, we input a pure quarter sine wave as a data vector and recover its Fourier quarter sine series.

```
      USE QSINF_INT
      USE CONST_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=7)
!
      INTEGER    I, NOUT
      REAL       COEF(N), FLOAT, PI, SIN, SEQ(N)
      INTRINSIC  FLOAT, SIN
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Fill the data vector SEQ
!                                 with a pure sine wave
      PI = CONST('PI')
      DO 10  I=1, N
         SEQ(I) = SIN(FLOAT(I)*(PI/2.0)/FLOAT(N))
   10 CONTINUE
!                                 Compute the transform of SEQ
      CALL QSINF (N, SEQ, COEF)
!                                 Print results
      WRITE (NOUT,99998)
      WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
99998 FORMAT (9X, 'INDEX', 6X, 'SEQ', 7X, 'COEF')
99999 FORMAT (1X, I11, 5X, F6.2, 5X, F6.2)
      END
```

Output

```
INDEX      SEQ       COEF
  1       0.22       7.00
  2       0.43       0.00
  3       0.62       0.00
  4       0.78       0.00
  5       0.90       0.00
  6       0.97       0.00
  7       1.00       0.00
```

Comments

1. Workspace may be explicitly provided, if desired, by use of Q2INF/DQ2INF. The reference is:

   ```
   CALL Q2INF (N, SEQ, COEF, WQSIN)
   ```

The additional argument is:

WQSIN — Array of length 3 * N + 15 initialized by QSINI (page 1037). The initialization depends on N. (Input)

2. The routine QSINF is most efficient when N is the product of small primes.

3. The arrays COEF and SEQ may be the same.

4. If QSINF/QSINB is used repeatedly with the same value of N, then call QSINI followed by repeated calls to Q2INF/Q2INB. This is more efficient than repeated calls to QSINF/QSINB.

Description

The routine QSINF computes the discrete Fourier quarter sine transform of a real vector of size *N*. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when *N* is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to *N* log *N*.

Specifically, given an N-vector *s* = SEQ, QSINF returns in *c* = COEF

$$c_m = 2\sum_{n=1}^{N-1} s_n \sin\left[\frac{(2m-1)n\pi}{2N}\right] + s_N(-1)^{m-1}$$

Finally, note that the Fourier quarter sine transform has an (unnormalized) inverse, which is implemented in the IMSL routine QSINB. The routine QSINF is based on the quarter sine FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

QSINB

Computes a sequence from its sine Fourier coefficients with only odd wave numbers.

Required Arguments

N — Length of the sequence to be transformed. (Input)

COEF — Array of length N containing the Fourier coefficients. (Input)

SEQ — Array of length N containing the sequence. (Output)

FORTRAN 90 Interface

Generic: CALL QSINB (N, COEF, SEQ)

Specific: The specific interface names are S_QSINB and D_QSINB.

FORTRAN 77 Interface

Single: CALL QSINB (N, COEF, SEQ)

Double: The double precision name is DQSINB.

Example

In this example, we first compute the quarter wave sine Fourier transform *c* of the vector *x* where $x_n = n$ for $n = 1$ to N. We then compute the inverse quarter wave Fourier transform of *c* which is $4Nx = s$.

```
      USE QSINB_INT
      USE QSINF_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=7)
!
      INTEGER    I, NOUT
      REAL       FLOAT, SEQ(N), COEF(N), X(N)
      INTRINSIC  FLOAT
!                                Get output unit number
      CALL UMACH (2, NOUT)
!                                Fill the data vector X
!                                with X(I) = I, I=1,N
      DO 10  I=1, N
         X(I) = FLOAT(I)
   10 CONTINUE
!                                Compute the forward transform of X
      CALL QSINF (N, X, COEF)
!                                Compute the backward transform of W
      CALL QSINB (N, COEF, SEQ)
!C                                Print results
      WRITE (NOUT,99998)
      WRITE (NOUT,99999) (X(I), COEF(I), SEQ(I), I=1,N)
99998 FORMAT (5X, 'INPUT', 5X, 'FORWARD TRANSFORM', 3X, 'BACKWARD ', &
            'TRANSFORM')
99999 FORMAT (3X, F6.2, 10X, F6.2, 15X, F6.2)
      END
```

Output

```
INPUT     FORWARD TRANSFORM   BACKWARD TRANSFORM
1.00           39.88                28.00
2.00           -4.58                56.00
3.00            1.77                84.00
4.00           -1.00               112.00
5.00            0.70               140.00
6.00           -0.56               168.00
7.00            0.51               196.00
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `Q2INB/DQ2INB`. The reference is:

   `CALL Q2INB (N, SEQ, COEF, WQSIN)`

   The additional argument is:

   WQSIN — ray of length 3 * N + 15 initialized by `QSINI` (page 1037). The initialization depends on N.(Input)

2. The routine `QSINB` is most efficient when `N` is the product of small primes.

3. The arrays `COEF` and `SEQ` may be the same.

4. If `QSINF`/`QSINB` is used repeatedly with the same value of `N`, then call `QSINI` followed by repeated calls to `Q2INF`/`Q2INB`. This is more efficient than repeated calls to `QSINF`/`QSINB`.

Description

The routine `QSINB` computes the discrete (unnormalized) inverse Fourier quarter sine transform of a real vector of size *N*. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when *N* is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to *N* log *N*.

Specifically, given an *N*-vector *c* = `COEF`, `QSINB` returns in *s* = `SEQ`

$$s_m = 4\sum_{n=1}^{N} c_n \sin\left(\frac{(2n-1)m\pi}{2N}\right)$$

Furthermore, a vector *x* of length *N* that is first transformed by `QSINF` (page 1032) and then by `QSINB` will be returned by `QSINB` as 4*Nx.* The routine `QSINB` is based on the inverse quarter sine FFT in FFTPACK which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

QSINI

Computes parameters needed by QSINF and QSINB.

```
CALL QSINI (N, WQSIN)
```

Required Arguments

N — Length of the sequence to be transformed. (Input)

WQSIN — Array of length 3N + 15 containing parameters needed by QSINF and QSINB. (Output)

FORTRAN 90 Interface

Generic: CALL QSINI (N, WQSIN)

Specific: The specific interface names are S_QSINI and D_QSINI.

FORTRAN 77 Interface

Single: CALL QSINI (N, WQSIN)

Double: The double precision name is DQSINI.

Example

In this example, we compute three distinct quarter sine transforms by calling QSINI once and then calling Q2INF three times.

```
      USE QSINI_INT
      USE CONST_INT
      USE Q2INF_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=7)
!
      INTEGER    I, K, NOUT
      REAL       COEF(N), FLOAT, PI, SIN, WQSIN(36), SEQ(N)
      INTRINSIC  FLOAT, SIN
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Initialize the work vector WQSIN
      CALL QSINI (N, WQSIN)
!                                 Different frequencies of the same
!                                 wave will be transformed
      PI = CONST('PI')
      DO 20  K=1, 3
!                                 Fill the data vector SEQ
!                                 with a pure sine wave
         DO 10  I=1, N
```

```
            SEQ(I) = SIN(FLOAT((2*K-1)*I)*(PI/2.0)/FLOAT(N))
   10    CONTINUE
!                                 Compute the transform of SEQ
         CALL Q2INF (N, SEQ, COEF, WQSIN)
!                                 Print results
         WRITE (NOUT,99998)
         WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
   20 CONTINUE
99998 FORMAT (/, 9X, 'INDEX', 6X, 'SEQ', 7X, 'COEF')
99999 FORMAT (1X, I11, 5X, F6.2, 5X, F6.2)
      END
```

Output

```
INDEX      SEQ       COEF
  1       0.22       7.00
  2       0.43       0.00
  3       0.62       0.00
  4       0.78       0.00
  5       0.90       0.00
  6       0.97       0.00
  7       1.00       0.00

INDEX      SEQ       COEF
  1       0.62       0.00
  2       0.97       7.00
  3       0.90       0.00
  4       0.43       0.00
  5      -0.22       0.00
  6      -0.78       0.00
  7      -1.00       0.00

INDEX      SEQ       COEF
  1       0.90       0.00
  2       0.78       0.00
  3      -0.22       7.00
  4      -0.97       0.00
  5      -0.62       0.00
  6       0.43       0.00
  7       1.00       0.00
```

Comments

Different WQSIN arrays are needed for different values of N.

Description

The routine QSINI initializes the routines QSINF (page 1032) and QSINB (page 1034). An efficient way to make multiple calls for the same *N* to IMSL routine QSINF or QSINB is to use routine QSINI for initialization. (In this case, replace QSINF or QSINB with Q2INF or Q2INB, respectively.) The routine QSINI is based on the routine SINQI in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

QCOSF

Computes the coefficients of the cosine Fourier transform with only odd wave numbers.

Required Arguments

N — Length of the sequence to be transformed. (Input)

SEQ — Array of length N containing the sequence. (Input)

COEF — Array of length N containing the Fourier coefficients. (Output)

FORTRAN 90 Interface

Generic: CALL QCOSF (N, SEQ, COEF [,…])

Specific: The specific interface names are S_QCOSF and D_QCOSF.

FORTRAN 77 Interface

Single: CALL QCOSF (N, SEQ, COEF)

Double: The double precision name is DQCOSF.

Example

In this example, we input a pure quarter cosine wave as a data vector and recover its Fourier quarter cosine series.

```
      USE QCOSF_INT
      USE CONST_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=7)
!
      INTEGER    I, NOUT
      REAL       COEF(N), COS, FLOAT, PI, SEQ(N)
      INTRINSIC  COS, FLOAT
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Fill the data vector SEQ
!                                 with a pure cosine wave
      PI = CONST('PI')
      DO 10  I=1, N
            SEQ(I) = COS(FLOAT(I-1)*(PI/2.0)/FLOAT(N))
   10    CONTINUE

!                                 Compute the transform of SEQ
         Call QCOSF (N, SEQ, COEF)
!                                  Print results
         WRITE (NOUT,99998)
         WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
99998 FORMAT (9X, 'INDEX', 6X, 'SEQ', 7X, 'COEF')
99999 FORMAT (1X, I11, 5X, F6.2, 5X, F6.2)
      END
```

Output

```
INDEX      SEQ        COEF
  1       1.00        7.00
  2       0.97        0.00
  3       0.90        0.00
  4       0.78        0.00
  5       0.62        0.00
  6       0.43        0.00
  7       0.22        0.00
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `Q2OSF/DQ2OSF`. The reference is:

   `CALL Q2OSF (N, SEQ, COEF, WQCOS)`

   The additional argument is:

   WQCOS — Array of length 3 * `N` + 15 initialized by `QCOSI` (page 1043). The initialization depends on `N`. (Input)

2. The routine `QCOSF` is most efficient when `N` is the product of small primes.

3. The arrays `COEF` and `SEQ` may be the same.

4. If `QCOSF/QCOSB` is used repeatedly with the same value of `N`, then call `QCOSI` followed by repeated calls to `Q2OSF/Q2OSB`. This is more efficient than repeated calls to `QCOSF/QCOSB`.

Description

The routine `QCOSF` computes the discrete Fourier quarter cosine transform of a real vector of size *N*. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when *N* is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to *N* log *N*.

Specifically, given an *N*-vector *s* = `SEQ`, `QCOSF` returns in *c* = `COEF`

$$c_m = s_1 + 2\sum_{n=2}^{N} s_n \cos\left(\frac{(2m-1)(n-1)\pi}{2N}\right)$$

Finally, note that the Fourier quarter cosine transform has an (unnormalized) inverse which is implemented in `QCOSB`. The routine `QCOSF` is based on the quarter cosine FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

QCOSB

Computes a sequence from its cosine Fourier coefficients with only odd wave numbers.

Required Arguments

N — Length of the sequence to be transformed. (Input)

COEF — Array of length N containing the Fourier coefficients. (Input)

SEQ — Array of length N containing the sequence. (Output)

FORTRAN 90 Interface

Generic: CALL QCOSB (N, COEF, SEQ)

Specific: The specific interface names are S_QCOSB and D_QCOSB.

FORTRAN 77 Interface

Single: CALL QCOSB (N, COEF, SEQ)

Double: The double precision name is DQCOSB.

Example

In this example, we first compute the quarter wave cosine Fourier transform c of the vector x, where $x_n = n$ for $n = 1$ to N. We then compute the inverse quarter wave Fourier transform of c which is $4Nx = s$.

```
      USE QCOSB_INT
      USE QCOSF_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=7)
!
      INTEGER    I, NOUT
      REAL       FLOAT, SEQ(N), COEF(N), X(N)
      INTRINSIC  FLOAT
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Fill the data vector X
!                                 with X(I) = I, I=1,N
      DO 10  I=1, N
         X(I) = FLOAT(I)
   10 CONTINUE
!                                 Compute the forward transform of X
      CALL QCOSF (N, X, COEF)
!                                 Compute the backward transform of
!                                 COEF
```

```
      CALL QCOSB (N, COEF, SEQ)
!                                 Print results
      WRITE (NOUT,99998)
      DO 20  I=1, N
         WRITE (NOUT,99999) X(I), COEF(I), SEQ(I)
   20 CONTINUE
99998 FORMAT (5X, 'INPUT', 5X, 'FORWARD TRANSFORM', 3X, 'BACKWARD ', &
            'TRANSFORM')
99999 FORMAT (3X, F6.2, 10X, F6.2, 15X, F6.2)
      END
```

Output

```
INPUT     FORWARD TRANSFORM   BACKWARD TRANSFORM
1.00           31.12                 28.00
2.00          -27.45                 56.00
3.00           10.97                 84.00
4.00           -9.00                112.00
5.00            4.33                140.00
6.00           -3.36                168.00
7.00            0.40                196.00
```

Comments

1. Workspace may be explicitly provided, if desired, by use of Q2OSB/DQ2OSB. The reference is:

   `CALL Q2OSB (N, COEF, SEQ, WQCOS)`

   The additional argument is:

   WQCOS — Array of length 3 * N + 15 initialized by QCOSI (page 1043). The initialization depends on N. (Input)

2. The routine QCOSB is most efficient when N is the product of small primes.

3. The arrays COEF and SEQ may be the same.

4. If QCOSF/QCOSB is used repeatedly with the same value of N, then call QCOSI followed by repeated calls to Q2OSF/Q2OSB. This is more efficient than repeated calls to QCOSF/QCOSB.

Description

The routine QCOSB computes the discrete (unnormalized) inverse Fourier quarter cosine transform of a real vector of size N. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when N is a product of small prime factors. If N satisfies this condition, then the computational effort is proportional to $N \log N$. Specifically, given an N-vector c = COEF, QCOSB returns in s = SEQ

$$s_m = 4\sum_{n=1}^{N} c_n \cos\left(\frac{(2n-1)(m-1)\pi}{2N}\right)$$

Furthermore, a vector *x* of length *N* that is first transformed by QCOSF (page 1039) and then by QCOSB will be returned by QCOSB as 4*Nx*. The routine QCOSB is based on the inverse quarter cosine FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

QCOSI

Computes parameters needed by QCOSF and QCOSB.

Required Arguments

N — Length of the sequence to be transformed. (Input)

WQCOS — Array of length 3N + 15 containing parameters needed by QCOSF and QCOSB. (Output)

FORTRAN 90 Interface

Generic: CALL QCOSI (N, WQCOS)

Specific: The specific interface names are S_QCOSI and D_QCOSI.

FORTRAN 77 Interface

Single: CALL QCOSI (N, WQCOS)

Double: The double precision name is DQCOSI.

Example

In this example, we compute three distinct quarter cosine transforms by calling QCOSI once and then calling Q2OSF three times.

```
      USE QCOSI_INT
      USE CONST_INT
      USE Q2OSF_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=7)
!
      INTEGER    I, K, NOUT
      REAL       COEF(N), COS, FLOAT, PI, WQCOS(36), SEQ(N)
      INTRINSIC  COS, FLOAT
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Initialize the work vector WQCOS
      CALL QCOSI (N, WQCOS)
!                                 Different frequencies of the same
!                                 wave will be transformed
      PI = CONST('PI')
      DO 20  K=1, 3
!                                  Fill the data vector SEQ
!                                  with a pure cosine wave
         DO 10  I=1, N
            SEQ(I) = COS(FLOAT((2*K-1)*(I-1))*(PI/2.0)/FLOAT(N))
   10    CONTINUE
!                                  Compute the transform of SEQ
         CALL Q2OSF (N, SEQ, COEF, WQCOS)
!                                  Print results
         WRITE (NOUT,99998)
         WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
   20 CONTINUE
99998 FORMAT (/, 9X, 'INDEX', 6X, 'SEQ', 7X, 'COEF')
99999 FORMAT (1X, I11, 5X, F6.2, 5X, F6.2)
      END
```

Output

```
INDEX      SEQ       COEF
  1       1.00       7.00
  2       0.97       0.00
  3       0.90       0.00
  4       0.78       0.00
  5       0.62       0.00
  6       0.43       0.00
  7       0.22       0.00

INDEX      SEQ       COEF
  1       1.00       0.00
  2       0.78       7.00
  3       0.22       0.00
  4      -0.43       0.00
  5      -0.90       0.00
  6      -0.97       0.00
  7      -0.62       0.00

INDEX      SEQ       COEF
  1       1.00       0.00
  2       0.43       0.00
  3      -0.62       7.00
  4      -0.97       0.00
  5      -0.22       0.00
  6       0.78       0.00
  7       0.90       0.00
```

Comments

Different WQCOS arrays are needed for different values of N.

Description

The routine QCOSI initializes the routines QCOSF (page 1039) and QCOSB (page 1041). An efficient way to make multiple calls for the same *N* to IMSL routine QCOSF or QCOSB is to use routine QCOSI for initialization. (In this case, replace QCOSF or QCOSB with Q2OSF or Q2OSB , respectively.) The routine QCOSI is based on the routine COSQI in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

FFT2D

Computes Fourier coefficients of a complex periodic two-dimensional array.

Required Arguments

A — NRA by NCA complex matrix containing the periodic data to be transformed. (Input)

COEF — NRA by NCA complex matrix containing the Fourier coefficients of A. (Output)

Optional Arguments

NRA — The number of rows of A. (Input)
Default: NRA = size (A,1).

NCA — The number of columns of A. (Input)
Default: NCA = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

LDCOEF — Leading dimension of COEF exactly as specified in the dimension statement of the calling program. (Input)
Default: LDCOEF = size (COEF,1).

FORTRAN 90 Interface

Generic: CALL FFT2D (A, COEF [,…])

Specific: The specific interface names are S_FFT2D and D_FFT2D.

FORTRAN 77 Interface

Single: CALL FFT2D (NRA, NCA, A, LDA, COEF, LDCOEF)

Double: The double precision name is DFFT2D.

Example

In this example, we compute the Fourier transform of the pure frequency input for a 5 × 4 array

$$a_{nm} = e^{2\pi i(n-1)2/N} e^{2\pi i(m-1)3/M}$$

for $1 \le n \le 5$ and $1 \le m \le 4$ using the IMSL routine FFT2D. The result

$$\hat{a} = c$$

has all zeros except in the (3, 4) position.

```
      USE FFT2D_INT
      USE CONST_INT
      USE WRCRN_INT
      INTEGER    I, IR, IS, J, NCA, NRA
      REAL       FLOAT, TWOPI
      COMPLEX    A(5,4), C, CEXP, CMPLX, COEF(5,4), H
      CHARACTER  TITLE1*26, TITLE2*26
      INTRINSIC  CEXP, CMPLX, FLOAT
!
      TITLE1 = 'The input matrix is below '
      TITLE2 = 'The output matrix is below'
      NRA    = 5
      NCA    = 4
      IR     = 3
      IS     = 4
!                                  Fill A with initial data
      TWOPI = CONST('PI')
      TWOPI = 2.0*TWOPI
      C     = CMPLX(0.0,1.0)
      H     = CEXP(TWOPI*C)
      DO 10  I=1, NRA
         DO 10  J=1, NCA
            A(I,J) = CEXP(TWOPI*C*((FLOAT((I-1)*(IR-1))/FLOAT(NRA)+ &
                    FLOAT((J-1)*(IS-1))/FLOAT(NCA))))
   10 CONTINUE
!
      CALL WRCRN (TITLE1, A)
!
      CALL FFT2D (A, COEF)
!
      CALL WRCRN (TITLE2, COEF)
!
      END
```

Output

```
                The input matrix is below
                1                2                3                4
1  ( 1.000, 0.000)  ( 0.000,-1.000)  (-1.000, 0.000)  ( 0.000, 1.000)
2  (-0.809, 0.588)  ( 0.588, 0.809)  ( 0.809,-0.588)  (-0.588,-0.809)
3  ( 0.309,-0.951)  (-0.951,-0.309)  (-0.309, 0.951)  ( 0.951, 0.309)
4  ( 0.309, 0.951)  ( 0.951,-0.309)  (-0.309,-0.951)  (-0.951, 0.309)
5  (-0.809,-0.588)  (-0.588, 0.809)  ( 0.809, 0.588)  ( 0.588,-0.809)

                 The Output matrix is below
                1                2                3                4
1  (  0.00,  0.00)  (  0.00,  0.00)  (  0.00,  0.00)  (  0.00,  0.00)
2  (  0.00,  0.00)  (  0.00,  0.00)  (  0.00,  0.00)  (  0.00,  0.00)
3  (  0.00,  0.00)  (  0.00,  0.00)  (  0.00,  0.00)  ( 20.00,  0.00)
4  (  0.00,  0.00)  (  0.00,  0.00)  (  0.00,  0.00)  (  0.00,  0.00)
5  (  0.00,  0.00)  (  0.00,  0.00)  (  0.00,  0.00)  (  0.00,  0.00)
```

Comments

1. Workspace may be explicitly provided, if desired, by use of F2T2D/DF2T2D. The reference is:

   ```
   CALL F2T2D (NRA, NCA, A, LDA, COEF, LDCOEF, WFF1,
          WFF2, CWK, CPY)
   ```

   The additional arguments are as follows:

   WFF1 — Real array of length 4 * NRA + 15 initialized by FFTCI. The initialization depends on NRA. (Input)

   WFF2 — Real array of length 4 * NCA + 15 initialized by FFTCI. The initialization depends on NCA. (Input)

   CWK — Complex array of length 1. (Workspace)

   CPY — Real array of length 2 * MAX(NRA, NCA). (Workspace)

2. The routine FFT2D is most efficient when NRA and NCA are the product of small primes.

3. The arrays COEF and A may be the same.

4. If FFT2D/FFT2B is used repeatedly, with the same values for NRA and NCA, then use FFTCI (page 1022) to fill WFF1(N = NRA) and WFF2(N = NCA). Follow this with repeated calls to F2T2D/F2T2B. This is more efficient than repeated calls to FFT2D/FFT2B.

Description

The routine FFT2D computes the discrete complex Fourier transform of a complex two dimensional array of size (NRA = N) × (NCA = M). The method used is a variant of the Cooley-Tukey algorithm , which is most efficient when N and M are each products of small prime factors. If N and M satisfy this condition, then the computational effort is proportional to $N M \log N M$. This considerable savings has historically led people to refer to this algorithm as the "fast Fourier transform" or FFT.

Specifically, given an $N \times M$ array a, FFT2D returns in c = COEF

$$c_{jk} = \sum_{n=1}^{N} \sum_{m=1}^{M} a_{nm} e^{-2\pi i(j-1)(n-1)/N} e^{-2\pi i(k-1)(m-1)/M}$$

Furthermore, a vector of Euclidean norm S is mapped into a vector of norm

$$\sqrt{NM}S$$

Finally, note that an unnormalized inverse is implemented in `FFT2B` (page 1048). The routine `FFT2D` is based on the complex FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

FFT2B

Computes the inverse Fourier transform of a complex periodic two-dimensional array.

Required Arguments

COEF — `NRCOEF` by `NCCOEF` complex array containing the Fourier coefficients to be transformed. (Input)

A — `NRCOEF` by `NCCOEF` complex array containing the Inverse Fourier coefficients of `COEF`. (Output)

Optional Arguments

NRCOEF — The number of rows of `COEF`. (Input)
Default: `NRCOEF` = size (`COEF`,1).

NCCOEF — The number of columns of `COEF`. (Input)
Default: `NCCOEF` = size (`COEF`,2).

LDCOEF — Leading dimension of `COEF` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDCOEF` = size (`COEF`,1).

LDA — Leading dimension of `A` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDA` = size (`A`,1).

FORTRAN 90 Interface

Generic: `CALL FFT2B (COEF, A [,…])`

Specific: The specific interface names are `S_FFT2B` and `D_FFT2B`.

FORTRAN 77 Interface

Single: `CALL FFT2B (NRCOEF, NCCOEF, COEF, LDCOEF, A, LDA)`

Double: The double precision name is `DFFT2B`.

Example

In this example, we first compute the Fourier transform of the 5 × 4 array

$$x_{nm} = n + 5\left(m - 1\right)$$

for $1 \le n \le 5$ and $1 \le m \le 4$ using the IMSL routine `FFT2D`. The result

$$\hat{x} = c$$

is then inverted by a call to `FFT2B`. Note that the result is an array a satisfying $a = (5)(4)x = 20x$. In general, `FFT2B` is an unnormalized inverse with expansion factor $N\,M$.

```
      USE FFT2B_INT
      USE FFT2D_INT
      USE WRCRN_INT
      INTEGER    M, N, NCA, NRA
      COMPLEX    CMPLX, X(5,4), A(5,4), COEF(5,4)
      CHARACTER  TITLE1*26, TITLE2*26, TITLE3*26
      INTRINSIC  CMPLX
!
      TITLE1 = ’The input matrix is below ’
      TITLE2 = ’After FFT2D               ’
      TITLE3 = ’After FFT2B               ’
      NRA    = 5
      NCA    = 4
!                                  Fill X with initial data
      DO 20  N=1, NRA
         DO 10  M=1, NCA
            X(N,M) = CMPLX(FLOAT(N+5*M-5),0.0)
   10    CONTINUE
   20 CONTINUE
!
      CALL WRCRN (TITLE1, X)
!
      CALL FFT2D (X, COEF)
!
      CALL WRCRN (TITLE2, COEF)
!
      CALL FFT2B (COEF, A)
!
      CALL WRCRN (TITLE3, A)
!
      END
```

Output

```
                The input matrix is below
                1                2                3                4
1  (  1.00,  0.00)  (  6.00,  0.00)  ( 11.00,  0.00)  ( 16.00,  0.00)
2  (  2.00,  0.00)  (  7.00,  0.00)  ( 12.00,  0.00)  ( 17.00,  0.00)
3  (  3.00,  0.00)  (  8.00,  0.00)  ( 13.00,  0.00)  ( 18.00,  0.00)
4  (  4.00,  0.00)  (  9.00,  0.00)  ( 14.00,  0.00)  ( 19.00,  0.00)
5  (  5.00,  0.00)  ( 10.00,  0.00)  ( 15.00,  0.00)  ( 20.00,  0.00)

                             After FFT2D
                1                2                3                4
1  ( 210.0,   0.0)  ( -50.0,  50.0)  ( -50.0,   0.0)  ( -50.0, -50.0)
2  ( -10.0,  13.8)  (   0.0,   0.0)  (   0.0,   0.0)  (   0.0,   0.0)
3  ( -10.0,   3.2)  (   0.0,   0.0)  (   0.0,   0.0)  (   0.0,   0.0)
```

```
4  ( -10.0,  -3.2)  (   0.0,   0.0)  (   0.0,   0.0)  (   0.0,   0.0)
5  ( -10.0, -13.8)  (   0.0,   0.0)  (   0.0,   0.0)  (   0.0,   0.0)

                             After FFT2B
                1                2                3                4
1  (  20.0,   0.0)  ( 120.0,   0.0)  ( 220.0,   0.0)  ( 320.0,   0.0)
2  (  40.0,   0.0)  ( 140.0,   0.0)  ( 240.0,   0.0)  ( 340.0,   0.0)
3  (  60.0,   0.0)  ( 160.0,   0.0)  ( 260.0,   0.0)  ( 360.0,   0.0)
4  (  80.0,   0.0)  ( 180.0,   0.0)  ( 280.0,   0.0)  ( 380.0,   0.0)
5  ( 100.0,   0.0)  ( 200.0,   0.0)  ( 300.0,   0.0)  ( 400.0,   0.0)
```

Comments

1. Workspace may be explicitly provided, if desired, by use of F2T2B/DF2T2B. The reference is:

   ```
   CALL F2T2B (NRCOEF, NCCOEF, A, LDA, COEF, LDCOEF,
   WFF1, WFF2, CWK, CPY)
   ```

   The additional arguments are as follows:

   WFF1 — Real array of length 4 * NRCOEF + 15 initialized by FFTCI (page 1022). The initialization depends on NRCOEF. (Input)

   WFF2 — Real array of length 4 * NCCOEF + 15 initialized by FFTCI. The initialization depends on NCCOEF. (Input)

   CWK — Complex array of length 1. (Workspace)

   CPY — Real array of length 2 * MAX(NRCOEF, NCCOEF). (Workspace)

2. The routine FFT2B is most efficient when NRCOEF and NCCOEF are the product of small primes.

3. The arrays COEF and A may be the same.

4. If FFT2D/FFT2B is used repeatedly, with the same values for NRCOEF and NCCOEF, then use FFTCI to fill WFF1(N = NRCOEF) and WFF2(N = NCCOEF). Follow this with repeated calls to F2T2D/F2T2B. This is more efficient than repeated calls to FFT2D/FFT2B.

Description

The routine FFT2B computes the inverse discrete complex Fourier transform of a complex two-dimensional array of size (NRCOEF = N) × (NCCOEF = M). The method used is a variant of the Cooley-Tukey algorithm , which is most efficient when N and M are both products of small prime factors. If N and M satisfy this condition, then the computational effort is proportional to $N M \log N M$. This considerable savings has historically led people to refer to this algorithm as the “fast Fourier transform” or FFT.

Specifically, given an $N \times M$ array c = COEF, FFT2B returns in a

$$a_{jk} = \sum_{n=1}^{N} \sum_{m=1}^{M} c_{nm} e^{2\pi i(j-1)(n-1)/N} e^{2\pi i(k-1)(m-1)/M}$$

Furthermore, a vector of Euclidean norm S is mapped into a vector of norm

$$S\sqrt{NM}$$

Finally, note that an unnormalized inverse is implemented in FFT2D (page 1045). The routine FFT2B is based on the complex FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

FFT3F

Computes Fourier coefficients of a complex periodic three-dimensional array.

Required Arguments

A — Three-dimensional complex matrix containing the data to be transformed. (Input)

B — Three-dimensional complex matrix containing the Fourier coefficients of A. (Output)
The matrices A and B may be the same.

Optional Arguments

N1 — Limit on the first subscript of matrices A and B. (Input)
Default: N1 = size(A, 1)

N2 — Limit on the second subscript of matrices A and B. (Input)
Default: N2 = size(A, 2)

N3 — Limit on the third subscript of matrices A and B. (Input)
Default: N3 = size(A, 3)

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

MDA — Middle dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: MDA = size (A,2).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

MDB — Middle dimension of `B` exactly as specified in the dimension statement of the calling program. (Input)
Default: `MDB` = size (`B`,2).

FORTRAN 90 Interface

Generic: `CALL FFT3F (A, B [,…])`

Specific: The specific interface names are `S_FFT3F` and `D_FFT3F`.

FORTRAN 77 Interface

Single: `CALL FFT3F (N1, N2, N3, A, LDA, MDA, B, LDB, MDB)`

Double: The double precision name is `DFFT3F`.

Example

In this example, we compute the Fourier transform of the pure frequency input for a 2 × 3 × 4 array

$$a_{nml} = e^{2\pi i(n-1)1/2} e^{2\pi i(m-1)2/3} e^{2\pi i(l-1)2/4}$$

for $1 \le n \le 2$, $1 \le m \le 3$, and $1 \le l \le 4$ using the IMSL routine `FFT3F`. The result

$$\hat{a} = c$$

has all zeros except in the (2, 3, 3) position.

```
      USE FFT3F_INT
      USE UMACH_INT
      USE CONST_INT
      INTEGER    LDA, LDB, MDA, MDB, NDA, NDB
      PARAMETER  (LDA=2, LDB=2, MDA=3, MDB=3, NDA=4, NDB=4)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    I, J, K, L, M, N, N1, N2, N3, NOUT
      REAL       PI
      COMPLEX    A(LDA,MDA,NDA), B(LDB,MDB,NDB), C, H
!                                 SPECIFICATIONS FOR INTRINSICS
      INTRINSIC  CEXP, CMPLX
      COMPLEX    CEXP, CMPLX
!                                 SPECIFICATIONS FOR SUBROUTINES
!                                 SPECIFICATIONS FOR FUNCTIONS
!                                 Get output unit number
      CALL UMACH (2, NOUT)
      PI = CONST('PI')
      C  = CMPLX(0.0,2.0*PI)
!                                 Set array A
      DO 30  N=1, 2
         DO 20  M=1, 3
            DO 10  L=1, 4
               H        = C*(N-1)*1/2 + C*(M-1)*2/3 + C*(L-1)*2/4
               A(N,M,L) = CEXP(H)
```

```
   10       CONTINUE
   20    CONTINUE
   30 CONTINUE
!
      CALL FFT3F (A, B)
!
      WRITE (NOUT,99996)
      DO 50  I=1, 2
         WRITE (NOUT,99998) I
         DO 40  J=1, 3
            WRITE (NOUT,99999) (A(I,J,K),K=1,4)
   40    CONTINUE
   50 CONTINUE
!
      WRITE (NOUT,99997)
      DO 70  I=1, 2
         WRITE (NOUT,99998) I
         DO 60  J=1, 3
            WRITE (NOUT,99999) (B(I,J,K),K=1,4)
   60    CONTINUE
   70 CONTINUE
!
99996 FORMAT (13X, ’The input for FFT3F is’)
99997 FORMAT (/, 13X, ’The results from FFT3F are’)
99998 FORMAT (/, ’ Face no. ’, I1)
99999 FORMAT (1X, 4(’(’,F6.2,’,’,F6.2,’)’,3X))
      END
```

Output

```
            The input for FFT3F is

Face no. 1
(  1.00,  0.00)   ( -1.00,  0.00)   (  1.00,  0.00)   ( -1.00,  0.00)
( -0.50, -0.87)   (  0.50,  0.87)   ( -0.50, -0.87)   (  0.50,  0.87)
( -0.50,  0.87)   (  0.50, -0.87)   ( -0.50,  0.87)   (  0.50, -0.87)

Face no. 2
( -1.00,  0.00)   (  1.00,  0.00)   ( -1.00,  0.00)   (  1.00,  0.00)
(  0.50,  0.87)   ( -0.50, -0.87)   (  0.50,  0.87)   ( -0.50, -0.87)
(  0.50, -0.87)   ( -0.50,  0.87)   (  0.50, -0.87)   ( -0.50,  0.87)

The results from FFT3F are

Face no. 1
(  0.00,  0.00)   (  0.00,  0.00)   (  0.00,  0.00)   (  0.00,  0.00)
(  0.00,  0.00)   (  0.00,  0.00)   (  0.00,  0.00)   (  0.00,  0.00)
(  0.00,  0.00)   (  0.00,  0.00)   (  0.00,  0.00)   (  0.00,  0.00)

Face no. 2
(  0.00,  0.00)   (  0.00,  0.00)   (  0.00,  0.00)   (  0.00,  0.00)
(  0.00,  0.00)   (  0.00,  0.00)   (  0.00,  0.00)   (  0.00,  0.00)
(  0.00,  0.00)   (  0.00,  0.00)   ( 24.00,  0.00)   (  0.00,  0.00)
```

Comments

1. Workspace may be explicitly provided, if desired, by use of F2T3F/DF2T3F. The reference is:

```
CALL F2T3F (N1, N2, N3, A, LDA, MDA, B, LDB, MDB,
WFF1, WFF2, WFF3, CPY)
```

The additional arguments are as follows:

WFF1 — Real array of length 4 * N1 + 15 initialized by FFTCI (page 1022). The initialization depends on N1. (Input)

WFF2 — Real array of length 4 * N2 + 15 initialized by FFTCI. The initialization depends on N2. (Input)

WFF3 — Real array of length 4 * N3 + 15 initialized by FFTCI. The initialization depends on N3. (Input)

CPY — Real array of size 2 * MAX(N1, N2, N3). (Workspace)

2. The routine FFT3F is most efficient when N1, N2, and N3 are the product of small primes.

3. If FFT3F/FFT3B is used repeatedly with the same values for N1, N2 and N3, then use FFTCI to fill WFF1(N = N1), WFF2(N = N2), and WFF3(N = N3). Follow this with repeated calls to F2T3F/F2T3B. This is more efficient than repeated calls to FFT3F/FFT3B.

Description

The routine FFT3F computes the forward discrete complex Fourier transform of a complex three-dimensional array of size (N1 = N) × (N2 = M) × (N3 = L). The method used is a variant of the Cooley-Tukey algorithm , which is most efficient when N, M, and L are each products of small prime factors. If N, M, and L satisfy this condition, then the computational effort is proportional to $N\,M\,L \log N\,M\,L$. This considerable savings has historically led people to refer to this algorithm as the “fast Fourier transform” or FFT.

Specifically, given an $N \times M \times L$ array a, FFT3F returns in c = COEF

$$c_{jkl} = \sum_{n=1}^{N} \sum_{m=1}^{M} \sum_{l=1}^{L} a_{nml} e^{-2\pi i (j-1)(n-1)/N} e^{-2\pi i (k-1)(m-1)/M} e^{-2\pi i (k-1)(l-1)/L}$$

Furthermore, a vector of Euclidean norm S is mapped into a vector of norm

$$\sqrt{NML}S$$

Finally, note that an unnormalized inverse is implemented in FFT3B. The routine FFT3F is based on the complex FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

FFT3B

Computes the inverse Fourier transform of a complex periodic three-dimensional array.

Required Arguments

A — Three-dimensional complex matrix containing the data to be transformed. (Input)

B — Three-dimensional complex matrix containing the inverse Fourier coefficients of A. (Output)
The matrices A and B may be the same.

Optional Arguments

N1 — Limit on the first subscript of matrices A and B. (Input)
Default: N1 = size (A,1).

N2 — Limit on the second subscript of matrices A and B. (Input)
Default: N2 = size (A,2).

N3 — Limit on the third subscript of matrices A and B. (Input)
Default: N3 = size (A,3).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

MDA — Middle dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: MDA = size (A,2).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

MDB — Middle dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: MDB = size (B,2).

FORTRAN 90 Interface

Generic: `CALL FFT3B (A, B [,…])`

Specific: The specific interface names are S_FFT3B and D_FFT3B.

FORTRAN 77 Interface

Single: `CALL FFT3B (N1, N2, N3, A, LDA, MDA, B, LDB, MDB)`

Double: The double precision name is `DFFT3B`.

Example

In this example, we compute the Fourier transform of the 2 × 3 × 4 array

$$x_{nml} = n + 2(m-1) + 2(3)(l-1)$$

for $1 \le n \le 2$, $1 \le m \le 3$, and $1 \le l \le 4$ using the IMSL routine `FFT3F`. The result

$$a = \hat{x}$$

is then inverted using `FFT3B`. Note that the result is an array b satisfying $b = 2(3)(4)x = 24x$. In general, `FFT3B` is an unnormalized inverse with expansion factor $N\,M\,L$.

```
      USE FFT3B_INT
      USE FFT3F_INT
      USE UMACH_INT
      INTEGER    LDA, LDB, MDA, MDB, NDA, NDB
      PARAMETER  (LDA=2, LDB=2, MDA=3, MDB=3, NDA=4, NDB=4)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    I, J, K, L, M, N, N1, N2, N3, NOUT
      COMPLEX    A(LDA,MDA,NDA), B(LDB,MDB,NDB), X(LDB,MDB,NDB)
!                                 SPECIFICATIONS FOR INTRINSICS
      INTRINSIC  CEXP, CMPLX
      COMPLEX    CEXP, CMPLX
!                                 SPECIFICATIONS FOR SUBROUTINES
!                                 Get output unit number
      CALL UMACH (2, NOUT)
      N1 = 2
      N2 = 3
      N3 = 4
!                                 Set array X
      DO 30  N=1, 2
         DO 20  M=1, 3
            DO 10  L=1, 4
               X(N,M,L) = N + 2*(M-1) + 2*3*(L-1)
   10       CONTINUE
   20    CONTINUE
   30 CONTINUE
!
      CALL FFT3F (X, A)
      CALL FFT3B (A, B)
!
      WRITE (NOUT,99996)
      DO 50  I=1, 2
         WRITE (NOUT,99998) I
         DO 40  J=1, 3
            WRITE (NOUT,99999) (X(I,J,K),K=1,4)
   40    CONTINUE
   50 CONTINUE
```

```
!
      WRITE (NOUT,99997)
      DO 70  I=1, 2
         WRITE (NOUT,99998) I
         DO 60  J=1, 3
            WRITE (NOUT,99999) (A(I,J,K),K=1,4)
   60    CONTINUE
   70 CONTINUE
!
      WRITE (NOUT, 99995)
      DO 90  I=1, 2
         WRITE (NOUT,99998) I
         DO 80  J=1, 3
            WRITE (NOUT,99999) (B(I,J,K),K=1,4)
   80    CONTINUE
   90 CONTINUE
99995 FORMAT (13X, ’The unnormalized inverse is’)
99996 FORMAT (13X, ’The input for FFT3F is’)
99997 FORMAT (/, 13X, ’The results from FFT3F are’)
99998 FORMAT (/, ’ Face no. ’, I1)
99999 FORMAT (1X, 4(’(’,F6.2,’,’,F6.2,’)’,3X))
      END
```

Output

```
            The input for FFT3F is

Face no. 1
(  1.00,  0.00)   (  7.00,  0.00)   ( 13.00,  0.00)   ( 19.00,  0.00)
(  3.00,  0.00)   (  9.00,  0.00)   ( 15.00,  0.00)   ( 21.00,  0.00)
(  5.00,  0.00)   ( 11.00,  0.00)   ( 17.00,  0.00)   ( 23.00,  0.00)

Face no. 2
(  2.00,  0.00)   (  8.00,  0.00)   ( 14.00,  0.00)   ( 20.00,  0.00)
(  4.00,  0.00)   ( 10.00,  0.00)   ( 16.00,  0.00)   ( 22.00,  0.00)
(  6.00,  0.00)   ( 12.00,  0.00)   ( 18.00,  0.00)   ( 24.00,  0.00)

The results from FFT3F are

Face no. 1
(300.00,  0.00)   (-72.00, 72.00)   (-72.00,  0.00)   (-72.00,-72.00)
(-24.00, 13.86)   (  0.00,  0.00)   (  0.00,  0.00)   (  0.00,  0.00)
(-24.00,-13.86)   (  0.00,  0.00)   (  0.00,  0.00)   (  0.00,  0.00)

Face no. 2
(-12.00,  0.00)   (  0.00,  0.00)   (  0.00,  0.00)   (  0.00,  0.00)
(  0.00,  0.00)   (  0.00,  0.00)   (  0.00,  0.00)   (  0.00,  0.00)
(  0.00,  0.00)   (  0.00,  0.00)   (  0.00,  0.00)   (  0.00,  0.00)

The unnormalized inverse is

Face no. 1
( 24.00,  0.00)   (168.00,  0.00)   (312.00,  0.00)   (456.00,  0.00)
( 72.00,  0.00)   (216.00,  0.00)   (360.00,  0.00)   (504.00,  0.00)
(120.00,  0.00)   (264.00,  0.00)   (408.00,  0.00)   (552.00,  0.00)
```

```
Face no. 2
( 48.00,  0.00)   (192.00,  0.00)   (336.00,  0.00)   (480.00,  0.00)
( 96.00,  0.00)   (240.00,  0.00)   (384.00,  0.00)   (528.00,  0.00)
(144.00,  0.00)   (288.00,  0.00)   (432.00,  0.00)   (576.00,  0.00)
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `F2T3B/DF2T3B`. The reference is:

   ```
   CALL F2T3B (N1, N2, N3, A, LDA, MDA, B, LDB, MDB,
   WFF1, WFF2, WFF3, CPY)
   ```

   The additional arguments are as follows:

   WFF1 — Real array of length 4 * `N1` + 15 initialized by `FFTCI` (page 1022). The initialization depends on `N1`. (Input)

   WFF2 — Real array of length 4 * `N2` + 15 initialized by `FFTCI`. The initialization depends on `N2`. (Input)

   WFF3 — Real array of length 4 * `N3` + 15 initialized by `FFTCI`. The initialization depends on `N3`. (Input)

   CPY — Real array of size 2 * `MAX(N1, N2, N3)`. (Workspace)

2. The routine `FFT3B` is most efficient when `N1`, `N2`, and `N3` are the product of small primes.

3. If `FFT3F/FFT3B` is used repeatedly with the same values for `N1`, `N2` and `N3`, then use `FFTCI` to fill `WFF1(N = N1)`, `WFF2(N = N2)`, and `WFF3(N = N3)`. Follow this with repeated calls to `F2T3F/F2T3B`. This is more efficient than repeated calls to `FFT3F/FFT3B`.

Description

The routine `FFT3B` computes the inverse discrete complex Fourier transform of a complex three-dimensional array of size (`N1` = *N*) × (`N2` = *M*) × (`N3` = *L*). The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when *N*, *M*, and *L* are each products of small prime factors. If *N*, *M*, and *L* satisfy this condition, then the computational effort is proportional to *N M L* log *N M L*. This considerable savings has historically led people to refer to this algorithm as the “fast Fourier transform” or FFT.

Specifically, given an *N* × *M* × *L* array *a*, `FFT3B` returns in *b*

$$b_{jkl}\sum_{n=1}^{N}\sum_{m=1}^{M}\sum_{l=1}^{L}a_{nml}e^{2\pi i(j-1)(n-1)/N}e^{2\pi i(k-1)(m-1)/M}e^{2\pi i(k-1)(l-1)/L}$$

Furthermore, a vector of Euclidean norm *S* is mapped into a vector of norm

$$\sqrt{NMLS}$$

Finally, note that an unnormalized inverse is implemented in `FFT3F`. The routine `FFT3B` is based on the complex FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

RCONV

Computes the convolution of two real vectors.

Required Arguments

X — Real vector of length `NX`. (Input)

Y — Real vector of length `NY`. (Input)

Z — Real vector of length `NZ` ontaining the convolution of `X` and `Y`. (Output)

ZHAT — Real vector of length `NZ` containing the discrete Fourier transform of `Z`. (Output)

Optional Arguments

IDO — Flag indicating the usage of `RCONV`. (Input)
Default: `IDO` = 0.

`IDO`	**Usage**
0	If this is the only call to `RCONV`.

If `RCONV` is called multiple times in sequence with the same `NX`, `NY`, and `IPAD`, `IDO` should be set to

1	on the first call
2	on the intermediate calls
3	on the final call.

NX — Length of the vector `X`. (Input)
Default: `NX` = size (`X`,1).

NY — Length of the vector `Y`. (Input)
Default: `NY` = size (`Y`,1).

IPAD — `IPAD` should be set to zero for periodic data or to one for nonperiodic data. (Input)
Default: `IPAD` = 0.

NZ — Length of the vector Z. (Input/Output)
Upon input: When IPAD is zero, NZ must be at least MAX(NX, NY). When IPAD is one, NZ must be greater than or equal to the smallest integer greater than or equal to (NX + NY −1) of the form $(2^{\alpha}) * (3^{\beta}) * (5^{\gamma})$ where alpha, beta, and gamma are nonnegative integers. Upon output, the value for NZ that was used by RCONV.
Default: NZ = size (Z,1).

FORTRAN 90 Interface

Generic: CALL RCONV (X, Y, Z, ZHAT [,…])

Specific: The specific interface names are S_RCONV and D_RCONV.

FORTRAN 77 Interface

Single: CALL RCONV (IDO, NX, X, NY, Y, IPAD, NZ, Z, ZHAT)

Double: The double precision name is DRCONV.

Example

In this example, we compute both a periodic and a non-periodic convolution. The idea here is that one can compute a moving average of the type found in digital filtering using this routine. The averaging operator in this case is especially simple and is given by averaging five consecutive points in the sequence. The periodic case tries to recover a noisy sin function by averaging five nearby values. The nonperiodic case tries to recover the values of an exponential function contaminated by noise. The large error for the last value printed has to do with the fact that the convolution is averaging the zeroes in the “pad” rather than function values. Notice that the signal size is 100, but we only report the errors at ten points.

```
      USE IMSL_LIBRARIES
      INTEGER    NFLTR, NY
      PARAMETER  (NFLTR=5, NY=100)
!
      INTEGER    I, IPAD, K, MOD, NOUT, NZ
      REAL       ABS, EXP, F1, F2, FLOAT, FLTR(NFLTR), &
                FLTRER, ORIGER, SIN, TOTAL1, TOTAL2, TWOPI, X, &
                Y(NY), Z(2*(NFLTR+NY-1)), ZHAT(2*(NFLTR+NY-1))
      INTRINSIC  ABS, EXP, FLOAT, MOD, SIN
!                                DEFINE FUNCTIONS
      F1(X) = SIN(X)
      F2(X) = EXP(X)
!
      CALL RNSET (1234579)
      CALL UMACH (2, NOUT)
      TWOPI = CONST('PI')
      TWOPI = 2.0*TWOPI
!                                 SET UP THE FILTER
      DO 10  I=1, 5
         FLTR(I) = 0.2
   10 CONTINUE
```

```
!                                 SET UP Y-VECTOR FOR THE PERIODIC
!                                 CASE.
      DO 20  I=1, NY
         X    = TWOPI*FLOAT(I-1)/FLOAT(NY-1)
         Y(I) = RNUNF()
         Y(I) = F1(X) + 0.5*Y(I) - 0.25
   20 CONTINUE
!                                 CALL THE CONVOLUTION ROUTINE FOR THE
!                                 PERIODIC CASE.
      NZ = 2*(NFLTR+NY-1)
      CALL RCONV (FLTR, Y, Z, ZHAT, IPAD=0, NZ=NZ)
!                                 PRINT RESULTS
      WRITE (NOUT,99993)
      WRITE (NOUT,99995)
      TOTAL1 = 0.0
      TOTAL2 = 0.0
      DO 30  I=1, NY
!                                 COMPUTE THE OFFSET FOR THE Z-VECTOR
         IF (I .GE. NY-1) THEN
            K = I - NY + 2
         ELSE
            K = I + 2
         END IF
!
         X      = TWOPI*FLOAT(I-1)/FLOAT(NY-1)
         ORIGER = ABS(Y(I)-F1(X))
         FLTRER = ABS(Z(K)-F1(X))
         IF (MOD(I,11) .EQ. 1) WRITE (NOUT,99997) X, F1(X), ORIGER, &
            FLTRER
         TOTAL1 = TOTAL1 + ORIGER
         TOTAL2 = TOTAL2 + FLTRER
   30 CONTINUE
      WRITE (NOUT,99998) TOTAL1/FLOAT(NY)
      WRITE (NOUT,99999) TOTAL2/FLOAT(NY)
!                                 SET UP Y-VECTOR FOR THE NONPERIODIC
!                                 CASE.
      DO 40  I=1, NY
         A    = FLOAT(I-1)/FLOAT(NY-1)
         Y(I) = RNUNF()
         Y(I) = F2(A) + 0.5*Y(I) - 0.25
   40 CONTINUE
!                                 CALL THE CONVOLUTION ROUTINE FOR THE
!                                 NONPERIODIC CASE.
      NZ = 2*(NFLTR+NY-1)
      CALL RCONV (FLTR, Y, Z, ZHAT, IPAD=1)
!                                 PRINT RESULTS
      WRITE (NOUT,99994)
      WRITE (NOUT,99996)
      TOTAL1 = 0.0
      TOTAL2 = 0.0
      DO 50  I=1, NY
         X      = FLOAT(I-1)/FLOAT(NY-1)
         ORIGER = ABS(Y(I)-F2(X))
         FLTRER = ABS(Z(I+2)-F2(X))
         IF (MOD(I,11) .EQ. 1) WRITE (NOUT,99997) X, F2(X), ORIGER, &
```

```
            FLTRER
         TOTAL1 = TOTAL1 + ORIGER
         TOTAL2 = TOTAL2 + FLTRER
   50 CONTINUE
      WRITE (NOUT,99998) TOTAL1/FLOAT(NY)
      WRITE (NOUT,99999) TOTAL2/FLOAT(NY)
99993 FORMAT (' Periodic Case')
99994 FORMAT (/,' Nonperiodic Case')
99995 FORMAT (8X, 'x', 9X, 'sin(x)', 6X, 'Original Error', 5X, &
            'Filtered Error')
99996 FORMAT (8X, 'x', 9X, 'exp(x)', 6X, 'Original Error', 5X, &
            'Filtered Error')
99997 FORMAT (1X, F10.4, F13.4, 2F18.4)
99998 FORMAT (' Average absolute error before filter:', F10.5)
99999 FORMAT (' Average absolute error after filter:', F11.5)
      END
```

Output

```
Periodic Case
    x         sin(x)      Original Error     Filtered Error
 0.0000       0.0000            0.0811             0.0587
 0.6981       0.6428            0.0226             0.0781
 1.3963       0.9848            0.1526             0.0529
 2.0944       0.8660            0.0959             0.0125
 2.7925       0.3420            0.1747             0.0292
 3.4907      -0.3420            0.1035             0.0238
 4.1888      -0.8660            0.0402             0.0595
 4.8869      -0.9848            0.0673             0.0798
 5.5851      -0.6428            0.1044             0.0074
 6.2832       0.0000            0.0154             0.0018
 Average absolute error before filter:   0.12481
 Average absolute error after filter:    0.04778

Nonperiodic Case
    x         exp(x)      Original Error     Filtered Error
 0.0000       1.0000            0.1476             0.3915
 0.1111       1.1175            0.0537             0.0326
 0.2222       1.2488            0.1278             0.0932
 0.3333       1.3956            0.1136             0.0987
 0.4444       1.5596            0.1617             0.0964
 0.5556       1.7429            0.0071             0.0662
 0.6667       1.9477            0.1248             0.0713
 0.7778       2.1766            0.1556             0.0158
 0.8889       2.4324            0.1529             0.0696
 1.0000       2.7183            0.2124             1.0562
 Average absolute error before filter:   0.12538
 Average absolute error after filter:    0.07764
```

Comments

1. Workspace may be explicitly provided, if desired, by use of R2ONV/DR2ONV. The reference is:

```
CALL R2ONV (IDO, NX, X, NY, Y, IPAD, NZ, Z, ZHAT
XWK, YWK, WK)
```

The additional arguments are as follows:

XWK — Real work array of length `NZ`.

YWK — Real work array of length `NZ`.

WK — Real work arrary of length 2 * `NZ` + 15.

2. Informational error

Type	Code	
4	1	The length of the vector `Z` must be large enough to hold the results. An acceptable length is returned in `NZ`.

Description

The routine `RCONV` computes the discrete convolution of two sequences x and y. More precisely, let n_x be the length of x and n_y denote the length of y. If a circular convolution is desired, then `IPAD` must be set to zero. We set

$$n_z := \max\{n_x, n_y\}$$

and we pad out the shorter vector with zeroes. Then, we compute

$$z_i = \sum_{j=1}^{n_z} x_{i-j+1} y_j$$

where the index on x is interpreted as a positive number between 1 and n_z, modulo n_z.

The technique used to compute the z_i's is based on the fact that the (complex discrete) Fourier transform maps convolution into multiplication. Thus, the Fourier transform of z is given by

$$\hat{z}(n) = \hat{x}(n)\hat{y}(n)$$

where

$$\hat{z}(n) = \sum_{m=1}^{n_z} z_m e^{-2\pi i(m-1)(n-1)/n_z}$$

The technique used here to compute the convolution is to take the discrete Fourier transform of x and y, multiply the results together component-wise, and then take the inverse transform of this product. It is very important to make sure that n_z is a product of small primes if `IPAD` is set to zero. If n_z is a product of small primes, then the computational effort will be proportional to $n_z \log(n_z)$. If `IPAD` is one, then a good value is chosen for n_z so that the Fourier transforms are efficient and $n_z \ge n_x + n_y - 1$. This will mean that both vectors will be padded with zeroes.

We point out that no complex transforms of x or y are taken since both sequences are real, we can take real transforms and simulate the complex transform above. This can produce a savings of a factor of six in time as well as save space over using the complex transform.

CCONV

Computes the convolution of two complex vectors.

Required Arguments

X — Complex vector of length NX. (Input)

Y — Complex vector of length NY. (Input)

Z — Complex vector of length NZ containing the convolution of X and Y. (Output)

ZHAT — Complex vector of length NZ containing the discrete complex Fourier transform of Z. (Output)

Optional Arguments

IDO — Flag indicating the usage of CCONV. (Input)
Default: IDO = 0.

IDO	Usage
0	If this is the only call to CCONV.

If CCONV is called multiple times in sequence with the same NX, NY, and IPAD, IDO should be set to:

1	on the first call
2	on the intermediate calls
3	on the final call.

NX — Length of the vector X. (Input)
Default: NX = size (X,1).

NY — Length of the vector Y. (Input)
Default: NY = size (Y,1).

IPAD — IPAD should be set to zero for periodic data or to one for nonperiodic data. (Input)
Default: IPAD =0.

NZ — Length of the vector Z. (Input/Output)
Upon input: When IPAD is zero, NZ must be at least MAX(NX, NY). When IPAD is one, NZ must be greater than or equal to the smallest integer greater than or equal to (NX + NY − 1) of the form $(2^\alpha) * (3^\beta) * (5^\gamma)$ where alpha, beta, and gamma are nonnegative

integers. Upon output, the value for NZ that was used by CCONV.
Default: NZ = size (Z,1).

FORTRAN 90 Interface

Generic: CALL CCONV (X, Y, Z, ZHAT [,…])

Specific: The specific interface names are S_CCONV and D_CCONV.

FORTRAN 77 Interface

Single: CALL CCONV (IDO, NX, X, NY, Y, IPAD, NZ, Z, ZHAT)

Double: The double precision name is DCCONV.

Example

In this example, we compute both a periodic and a non-periodic convolution. The idea here is that one can compute a moving average of the type found in digital filtering using this routine. The averaging operator in this case is especially simple and is given by averaging five consecutive points in the sequence. The periodic case tries to recover a noisy function $f_1(x) = \cos(x) + i\sin(x)$ by averaging five nearby values. The nonperiodic case tries to recover the values of the function $f_2(x) = e^x f_1(x)$ contaminated by noise. The large error for the first and last value printed has to do with the fact that the convolution is averaging the zeroes in the "pad" rather than function values. Notice that the signal size is 100, but we only report the errors at ten points.

```
      USE IMSL_LIBRARIES
      INTEGER    NFLTR, NY
      PARAMETER  (NFLTR=5, NY=100)
!
      INTEGER    I, IPAD, K, MOD, NOUT, NZ
      REAL       CABS, COS, EXP, FLOAT, FLTRER, ORIGER,  &
                SIN, TOTAL1, TOTAL2, TWOPI, X, T1, T2
      COMPLEX    CMPLX, F1, F2, FLTR(NFLTR), Y(NY), Z(2*(NFLTR+NY-1)), &
                ZHAT(2*(NFLTR+NY-1))
      INTRINSIC  CABS, CMPLX, COS, EXP, FLOAT, MOD, SIN
!                                DEFINE FUNCTIONS
      F1(X) = CMPLX(COS(X),SIN(X))
      F2(X) = EXP(X)*CMPLX(COS(X),SIN(X))
!
      CALL RNSET (1234579)
      CALL UMACH (2, NOUT)
      TWOPI = CONST('PI')
      TWOPI = 2.0*TWOPI
!                                 SET UP THE FILTER
      CALL CSET(NFLTR,(0.2,0.0),FLTR,1)
!                                 SET UP Y-VECTOR FOR THE PERIODIC
!                                 CASE.
      DO 20  I=1, NY
         X    = TWOPI*FLOAT(I-1)/FLOAT(NY-1)
         T1   = RNUNF()
         T2   = RNUNF()
         Y(I) = F1(X) + CMPLX(0.5*T1-0.25,0.5*T2-0.25)
   20 CONTINUE
!                                 CALL THE CONVOLUTION ROUTINE FOR THE
!                                 PERIODIC CASE.
      NZ = 2*(NFLTR+NY-1)
      CALL CCONV (FLTR, Y, Z, ZHAT)
!                                 PRINT RESULTS
      WRITE (NOUT,99993)
      WRITE (NOUT,99995)
      TOTAL1 = 0.0
      TOTAL2 = 0.0
      DO 30  I=1, NY
!                                 COMPUTE THE OFFSET FOR THE Z-VECTOR
         IF (I .GE. NY-1) THEN
            K = I - NY + 2
         ELSE
            K = I + 2
         END IF
!
         X      = TWOPI*FLOAT(I-1)/FLOAT(NY-1)
         ORIGER = CABS(Y(I)-F1(X))
         FLTRER = CABS(Z(K)-F1(X))
         IF (MOD(I,11) .EQ. 1) WRITE (NOUT,99997) X, F1(X), ORIGER, &
            FLTRER
         TOTAL1 = TOTAL1 + ORIGER
         TOTAL2 = TOTAL2 + FLTRER
   30 CONTINUE
      WRITE (NOUT,99998) TOTAL1/FLOAT(NY)
      WRITE (NOUT,99999) TOTAL2/FLOAT(NY)
!                                 SET UP Y-VECTOR FOR THE NONPERIODIC
!                                 CASE.
      DO 40  I=1, NY
         X    = FLOAT(I-1)/FLOAT(NY-1)
         T1   = RNUNF()
         T2   = RNUNF()
         Y(I) = F2(X) + CMPLX(0.5*T1-0.25,0.5*T2-0.25)
   40 CONTINUE
!                                 CALL THE CONVOLUTION ROUTINE FOR THE
!                                 NONPERIODIC CASE.
      NZ = 2*(NFLTR+NY-1)
      CALL CCONV (FLTR, Y, Z, ZHAT, IPAD=1)
!                                 PRINT RESULTS
      WRITE (NOUT,99994)
      WRITE (NOUT,99996)
      TOTAL1 = 0.0
      TOTAL2 = 0.0
      DO 50  I=1, NY
         X      = FLOAT(I-1)/FLOAT(NY-1)
         ORIGER = CABS(Y(I)-F2(X))
         FLTRER = CABS(Z(I+2)-F2(X))
         IF (MOD(I,11) .EQ. 1) WRITE (NOUT,99997) X, F2(X), ORIGER, &
            FLTRER
         TOTAL1 = TOTAL1 + ORIGER
         TOTAL2 = TOTAL2 + FLTRER
   50 CONTINUE
      WRITE (NOUT,99998) TOTAL1/FLOAT(NY)
      WRITE (NOUT,99999) TOTAL2/FLOAT(NY)
99993 FORMAT (' Periodic Case')
99994 FORMAT (/, ' Nonperiodic Case')
99995 FORMAT (8X, 'x', 15X, 'f1(x)', 8X, 'Original Error', 5X, &
           'Filtered Error')
99996 FORMAT (8X, 'x', 15X, 'f2(x)', 8X, 'Original Error', 5X, &
           'Filtered Error')
99997 FORMAT (1X, F10.4, 5X, '(', F7.4, ',', F8.4, ' )', 5X, F8.4, &
           10X, F8.4)
99998 FORMAT (' Average absolute error before filter:', F11.5)
99999 FORMAT (' Average absolute error after filter:', F12.5)
      END
```

Output

```
Periodic Case
 x               f1(x)        Original Error     Filtered Error
 0.0000     ( 1.0000,  0.0000 )       0.1666            0.0773
 0.6981     ( 0.7660,  0.6428 )       0.1685            0.1399
 1.3963     ( 0.1736,  0.9848 )       0.1756            0.0368
 2.0944     (-0.5000,  0.8660 )       0.2171            0.0142
 2.7925     (-0.9397,  0.3420 )       0.1147            0.0200
 3.4907     (-0.9397, -0.3420 )       0.0998            0.0331
 4.1888     (-0.5000, -0.8660 )       0.1137            0.0586
 4.8869     ( 0.1736, -0.9848 )       0.2217            0.0843
 5.5851     ( 0.7660, -0.6428 )       0.1831            0.0744
 6.2832     ( 1.0000,  0.0000 )       0.3234            0.0893
 Average absolute error before filter:    0.19315
 Average absolute error after filter:     0.08296

Nonperiodic Case
 x               f2(x)        Original Error     Filtered Error
 0.0000     ( 1.0000,  0.0000 )       0.0783            0.4336
 0.1111     ( 1.1106,  0.1239 )       0.2434            0.0477
 0.2222     ( 1.2181,  0.2752 )       0.1819            0.0584
 0.3333     ( 1.3188,  0.4566 )       0.0703            0.1267
 0.4444     ( 1.4081,  0.6706 )       0.1458            0.0868
 0.5556     ( 1.4808,  0.9192 )       0.1946            0.0930
 0.6667     ( 1.5307,  1.2044 )       0.1458            0.0734
 0.7778     ( 1.5508,  1.5273 )       0.1815            0.0690
 0.8889     ( 1.5331,  1.8885 )       0.0805            0.0193
 1.0000     ( 1.4687,  2.2874 )       0.2396            1.1708
 Average absolute error before filter:    0.18549
 Average absolute error after filter:     0.09636
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `C2ONV/DC2ONV`. The reference is:

   ```
   CALL C2ONV (IDO, NX, X, NY, Y, IPAD, NZ, Z, ZHAT,
   XWK, YWK, WK)
   ```

The additional arguments are as follows:

XWK — Complex work array of length `NZ`.

YWK — Complex work array of length `NZ`.

WK — Real work array of length 6 * `NZ` + 15.

2. Informational error

Type	Code	
4	1	The length of the vector `Z` must be large enough to hold the results. An acceptable length is returned in `NZ`.

Description

The subroutine `CCONV` computes the discrete convolution of two complex sequences x and y. More precisely, let n_x be the length of x and n_y denote the length of y. If a circular convolution is desired, then `IPAD` must be set to zero. We set

$$n_z := \max\{n_x, n_y\}$$

and we pad out the shorter vector with zeroes. Then, we compute

$$z_i = \sum_{j=1}^{n_z} x_{i-j+1}\, y_j$$

where the index on x is interpreted as a positive number between 1 and n_z, modulo n_z.

The technique used to compute the z_i's is based on the fact that the (complex discrete) Fourier transform maps convolution into multiplication. Thus, the Fourier transform of z is given by

$$\hat{z}(n) = \hat{x}(n)\,\hat{y}(n)$$

where

$$\hat{z}(n) = \sum_{m=1}^{n_z} z_m e^{-2\pi i(m-1)(n-1)/n_z}$$

The technique used here to compute the convolution is to take the discrete Fourier transform of x and y, multiply the results together component-wise, and then take the inverse transform of this product. It is very important to make sure that n_z is a product of small primes if `IPAD` is set to zero. If n_z is a product of small primes, then the computational effort will be proportional to $n_z \log(n_z)$. If `IPAD` is one, then a a good value is chosen for n_z so that the Fourier transforms are efficient and $n_z \geq n_x + n_y - 1$. This will mean that both vectors will be padded with zeroes.

RCORL

Computes the correlation of two real vectors.

Required Arguments

X — Real vector of length N. (Input)

Y — Real vector of length N. (Input)

Z — Real vector of length NZ containing the correlation of X and Y. (Output)

ZHAT — Real vector of length NZ containing the discrete Fourier transform of Z. (Output)

Optional Arguments

IDO — Flag indicating the usage of RCORL. (Input)
Default: IDO = 0.

IDO	Usage
0	If this is the only call to RCORL.

If RCORL is called multiple times in sequence with the same NX, NY, and IPAD, IDO should be set to:

1	on the first call
2	on the intermediate calls
3	on the final call.

N — Length of the X and Y vectors. (Input)
Default: N = size (X,1).

IPAD — IPAD should be set as follows. (Input)
Default: IPAD = 0.

IPAD Value

IPAD 0 for periodic data with X and Y different.

IPAD 1 for nonperiodic data with X and Y different.

IPAD 2 for periodic data with X and Y identical.

IPAD 3 for nonperiodic data with X and Y identical.

NZ — Length of the vector Z. (Input/Output)
Upon input: When IPAD is zero or two, NZ must be at least (2 * N – 1). When IPAD is one or three, NZ must be greater than or equal to the smallest integer greater than or equal to (2 * N – 1) of the form (2^{α}) * (3^{β}) * (5^{γ}) where alpha, beta, and gamma are

nonnegative integers. Upon output, the value for NZ that was used by RCORL.
Default: NZ = size (Z,1).

FORTRAN 90 Interface

Generic: CALL RCORL (X, Y, Z, ZHAT [,…])

Specific: The specific interface names are S_RCORL and D_RCORL.

FORTRAN 77 Interface

Single: CALL RCORL (IDO, N, X, Y, IPAD, NZ, Z, ZHAT)

Double: The double precision name is DRCORL.

Example

In this example, we compute both a periodic and a non-periodic correlation between two distinct signals *x* and *y*. In the first case we have 100 equally spaced points on the interval [0, 2π] and $f_1(x) = \sin(x)$. We define *x* and *y* as follows

$$\begin{aligned} x_i &= f_1(2\pi \frac{i-1}{n-1}) & i &= 1, \ldots, n \\ y_i &= f_1(2\pi \frac{i-1}{n-1} + \frac{\pi}{2}) & i &= 1, \ldots, n \end{aligned}$$

Note that the maximum value of *z* (the correlation of *x* with *y*) occurs at *i* = 26, which corresponds to the offset.

The nonperiodic case uses the function $f_2(x) = \sin(x^2)$. The two input signals are on the interval [0, 4π].

$$\begin{aligned} x_i &= f_2(4\pi \frac{i-1}{n-1}) & i &= 1, \ldots, n \\ y_i &= f_2(4\pi \frac{i-1}{n-1} + \pi) & i &= 1, \ldots, n \end{aligned}$$

The offset of *x* to *y* is again (roughly) 26 and this is where *z* has its maximum value.

```
      USE IMSL_LIBRARIES
      INTEGER    N
      PARAMETER  (N=100)
!
      INTEGER    I, IPAD, K, NOUT, NZ
      REAL       A, F1, F2, FLOAT, PI, SIN, X(N), XNORM, &
                Y(N), YNORM, Z(4*N), ZHAT(4*N)
      INTRINSIC  FLOAT, SIN
!                                 Define functions
      F1(A) = SIN(A)
      F2(A) = SIN(A*A)
!
```

```
      CALL UMACH (2, NOUT)
      PI = CONST('pi')
!                                 Set up the vectors for the
!                                 periodic case.
      DO 10  I=1, N
         X(I) = F1(2.0*PI*FLOAT(I-1)/FLOAT(N-1))
         Y(I) = F1(2.0*PI*FLOAT(I-1)/FLOAT(N-1)+PI/2.0)
   10 CONTINUE
!                                 Call the correlation routine for the
!                                 periodic case.
      NZ = 2*N
      CALL RCORL (X, Y, Z, ZHAT)
!                                 Find the element of Z with the
!                                 largest normalized value.
      XNORM = SNRM2(N,X,1)
      YNORM = SNRM2(N,Y,1)
      DO 20  I=1, N
         Z(I) = Z(I)/(XNORM*YNORM)
   20 CONTINUE
      K = ISMAX(N,Z,1)
!                                 Print results for the periodic
!                                 case.
      WRITE (NOUT,99995)
      WRITE (NOUT,99994)
      WRITE (NOUT,99997)
      WRITE (NOUT,99998) K
      WRITE (NOUT,99999) K, Z(K)
!                                 Set up the vectors for the
!                                 nonperiodic case.
      DO 30  I=1, N
         X(I) = F2(4.0*PI*FLOAT(I-1)/FLOAT(N-1))
         Y(I) = F2(4.0*PI*FLOAT(I-1)/FLOAT(N-1)+PI)
   30 CONTINUE
!                                 Call the correlation routine for the
!                                 nonperiodic case.
      NZ = 4*N
      CALL RCORL (X, Y, Z, ZHAT, IPAD=1)
!                                 Find the element of Z with the
!                                 largest normalized value.
      XNORM = SNRM2(N,X,1)
      YNORM = SNRM2(N,Y,1)
      DO 40  I=1, N
         Z(I) = Z(I)/(XNORM*YNORM)
   40 CONTINUE
      K = ISMAX(N,Z,1)
!                                 Print results for the nonperiodic
!                                 case.
      WRITE (NOUT,99996)
      WRITE (NOUT,99994)
      WRITE (NOUT,99997)
      WRITE (NOUT,99998) K
      WRITE (NOUT,99999) K, Z(K)
99994 FORMAT (1X, 28('-'))
99995 FORMAT (' Case #1: Periodic data')
99996 FORMAT (/, ' Case #2: Nonperiodic data')
```

```
99997 FORMAT (’ The element of Z with the largest normalized ’)
99998 FORMAT (’ value is Z(’, I2, ’).’)
99999 FORMAT (’ The normalized value of Z(’, I2, ’) is’, F6.3)
      END
```

Output

```
Example #1: Periodic case
---------------------------
The element of Z with the largest normalized value is Z(26).
The normalized value of Z(26) is 1.000

Example #2: Nonperiodic case
---------------------------
The element of Z with the largest normalized value is Z(26).
The normalized value of Z(26) is 0.661
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `R2ORL/DR2ORL`. The reference is:

   ```
   CALL R2ORL (IDO, N, X, Y, IPAD, NZ, Z, ZHAT, XWK,
   YWK, WK)
   ```

   The additional arguments are as follows:

   XWK — Real work array of length NZ.

   YWK — Real work array of length NZ.

   WK — Real work arrary of length 2 * NZ + 15.

2. Informational error

Type	Code	
4	1	The length of the vector `Z` must be large enough to hold the results. An acceptable length is returned in `NZ`.

Description

The subroutine `RCORL` computes the discrete correlation of two sequences *x* and *y*. More precisely, let *n* be the length of *x* and *y*. If a circular correlation is desired, then `IPAD` must be set to zero (for *x* and *y* distinct) and two (for *x* = *y*). We set (on output)

$$n_z = n \qquad \text{if IPAD} = 0, 2$$

$$n_z = 2^\alpha 3^\beta 5^\gamma \geq 2n - 1 \qquad \text{if IPAD} = 1, 3$$

where α, β, γ are nonnegative integers yielding the smallest number of the type $2^\alpha 3^\beta 5^\gamma$ satisfying the inequality. Once n_z is determined, we pad out the vectors with zeroes. Then, we compute

$$z_i = \sum_{j=1}^{n_z} x_{i+j-1} y_j$$

where the index on x is interpreted as a positive number between one and n_z, modulo n_z. Note that this means that

$$z_{n_z - k}$$

contains the correlation of $x(\cdot - \mathrm{k} - 1)$ with y as $k = 0, 1, \ldots, n_z/2$. Thus, if $x(k - 1) = y(k)$ for all k, then we would expect

$$z_{n_z}$$

to be the largest component of z.

The technique used to compute the z_i's is based on the fact that the (complex discrete) Fourier transform maps correlation into multiplication. Thus, the Fourier transform of z is given by

$$\hat{z}_j = \hat{x}_j \overline{\hat{y}}_j$$

where

$$\hat{z}_j \quad = \sum_{m=1}^{n_z} z_m e^{-2\pi i (m-1)(j-1)/n_z}$$

Thus, the technique used here to compute the correlation is to take the discrete Fourier transform of x and the conjugate of the discrete Fourier transform of y, multiply the results together component-wise, and then take the inverse transform of this product. It is very important to make sure that n_z is a product of small primes if `IPAD` is set to zero or two. If n_z is a product of small primes, then the computational effort will be proportional to $n_z \log(n_z)$. If `IPAD` is one or three, then a good value is chosen for n_z so that the Fourier transforms are efficient and $n_z \geq 2n - 1$. This will mean that both vectors will be padded with zeroes.

We point out that no complex transforms of x or y are taken since both sequences are real, and we can take real transforms and simulate the complex transform above. This can produce a savings of a factor of six in time as well as save space over using the complex transform.

CCORL

Computes the correlation of two complex vectors.

Required Arguments

X — Complex vector of length `N`. (Input)

Y — Complex vector of length `N`. (Input)

Z — Complex vector of length `NZ` containing the correlation of `X` and `Y`. (Output)

ZHAT — Complex vector of length `NZ` containing the inverse discrete complex Fourier transform of `Z`. (Output)

Optional Arguments

IDO — Flag indicating the usage of CCORL. (Input)
Default: IDO = 0.

IDO	Usage
0	If this is the only call to CCORL.

If CCORL is called multiple times in sequence with the same NX, NY, and IPAD, IDO should be set to:

1	on the first call
2	on the intermediate calls
3	on the final call.

N — Length of the X and Y vectors. (Input)
Default: N = size (X,1).

IPAD — IPAD should be set as follows. (Input)
IPAD = 0 for periodic data with X and Y different. IPAD = 1 for nonperiodic data with X and Y different. IPAD = 2 for periodic data with X and Y identical. IPAD = 3 for nonperiodic data with X and Y identical.
Default: IPAD = 0.

NZ — Length of the vector Z. (Input/Output)
Upon input: When IPAD is zero or two, NZ must be at least (2 * N – 1). When IPAD is one or three, NZ must be greater than or equal to the smallest integer greater than or equal to (2 * N – 1) of the form $(2^\alpha) * (3^\beta) * (5^\gamma)$ where alpha, beta, and gamma are nonnegative integers. Upon output, the value for NZ that was used by CCORL.
Default: NZ = size (Z,1).

FORTRAN 90 Interface

Generic: `CALL CCORL (X, Y, Z, ZHAT [,…])`

Specific: The specific interface names are S_CCORL and D_CCORL.

FORTRAN 77 Interface

Single: `CALL CCORL (IDO, N, X, Y, IPAD, NZ, Z, ZHAT)`

Double: The double precision name is DCCORL.

Example

In this example, we compute both a periodic and a non-periodic correlation between two distinct signals x and y. In the first case, we have 100 equally spaced points on the interval [0, 2π] and $f_1(x) = \cos(x) + i\sin(x)$. We define x and y as follows

$$\begin{aligned} x_i &= f_1(2\pi \frac{i-1}{n-1}) & i = 1, \ldots, n \\ y_i &= f_1(2\pi \frac{i-1}{n-1} + \frac{\pi}{2}) & i = 1, \ldots, n \end{aligned}$$

Note that the maximum value of z (the correlation of x with y) occurs at $i = 26$, which corresponds to the offset.

The nonperiodic case uses the function $f_2(x) = \cos(x^2) + i\sin(x^2)$. The two input signals are on the interval [0, 4π].

$$\begin{aligned} x_i &= f_2(4\pi \frac{i-1}{n-1}) & i = 1, \ldots, n \\ y_i &= f_2(4\pi \frac{i-1}{n-1} + \pi) & i = 1, \ldots, n \end{aligned}$$

The offset of x to y is again (roughly) 26 and this is where z has its maximum value.

```
      USE IMSL_LIBRARIES
      INTEGER    N
      PARAMETER  (N=100)
!
      INTEGER    I, IPAD, K, NOUT, NZ
      REAL       A, COS, F1, F2, FLOAT, PI, SIN, &
                XNORM, YNORM, ZREAL1(4*N)
      COMPLEX    CMPLX, X(N), Y(N), Z(4*N), ZHAT(4*N)
      INTRINSIC  CMPLX, COS, FLOAT, SIN
!                                Define functions
      F1(A) = CMPLX(COS(A),SIN(A))
      F2(A) = CMPLX(COS(A*A),SIN(A*A))
!
      CALL RNSET (1234579)
      CALL UMACH (2, NOUT)
      PI = CONST('pi')
!                                 Set up the vectors for the
!                                 periodic case.
      DO 10  I=1, N
         X(I) = F1(2.0*PI*FLOAT(I-1)/FLOAT(N-1))
         Y(I) = F1(2.0*PI*FLOAT(I-1)/FLOAT(N-1)+PI/2.0)
   10 CONTINUE
!                                 Call the correlation routine for the
!                                 periodic case.
      NZ = 2*N
      CALL CCORL (X, Y, Z, ZHAT, IPAD=0, NZ=NZ)
!                                 Find the element of Z with the
!                                 largest normalized real part.
      XNORM = SCNRM2(N,X,1)
      YNORM = SCNRM2(N,Y,1)
      DO 20  I=1, N
         ZREAL1(I) = REAL(Z(I))/(XNORM*YNORM)
   20 CONTINUE
      K = ISMAX(N,ZREAL1,1)
!                                 Print results for the periodic
!                                 case.
      WRITE (NOUT,99995)
      WRITE (NOUT,99994)
      WRITE (NOUT,99997)
      WRITE (NOUT,99998) K
      WRITE (NOUT,99999) K, ZREAL1(K)
!                                 Set up the vectors for the
!                                 nonperioddic case.
      DO 30  I=1, N
         X(I) = F2(4.0*PI*FLOAT(I-1)/FLOAT(N-1))
         Y(I) = F2(4.0*PI*FLOAT(I-1)/FLOAT(N-1)+PI)
   30 CONTINUE
!                                 Call the correlation routine for the
!                                 nonperiodic case.
      NZ = 4*N
      CALL CCORL (X, Y, Z, ZHAT, IPAD=1, NZ=NZ)
!                                 Find the element of z with the
!                                 largest normalized real part.
      XNORM = SCNRM2(N,X,1)
      YNORM = SCNRM2(N,Y,1)
      DO 40  I=1, N
         ZREAL1(I) = REAL(Z(I))/(XNORM*YNORM)
   40 CONTINUE
      K = ISMAX(N,ZREAL1,1)
!                                 Print results for the nonperiodic
!                                 case.
      WRITE (NOUT,99996)
      WRITE (NOUT,99994)
      WRITE (NOUT,99997)
      WRITE (NOUT,99998) K
      WRITE (NOUT,99999) K, ZREAL1(K)
99994 FORMAT (1X, 28(’-’))
99995 FORMAT (’ Case #1: periodic data’)
99996 FORMAT (/, ’ Case #2: nonperiodic data’)
99997 FORMAT (’ The element of Z with the largest normalized ’)
99998 FORMAT (’ real part is Z(’, I2, ’).’)
99999 FORMAT (’ The normalized value of real(Z(’, I2, ’)) is’, F6.3)
      END
```

Output

```
Example #1: periodic case
----------------------------
The element of Z with the largest normalized real part is Z(26).
The normalized value of real(Z(26)) is 1.000

Example #2: nonperiodic case
----------------------------
The element of Z with the largest normalized real part is Z(26).
The normalized value of real(Z(26)) is 0.638
```

Comments

1. Workspace may be explicitly provided, if desired, by use of C2ORL/DC2ORL. The reference is:

   ```
   CALL C2ORL (IDO, N, X, Y, IPAD, NZ, Z, ZHAT, XWK,
   YWK, WK)
   ```

   The additional arguments are as follows:

   XWK — Complex work array of length NZ.

   YWK — Complex work array of length NZ.

   WK — Real work arrary of length 6 * NZ + 15.

2. Informational error

Type	Code	
4	1	The length of the vector Z must be large enough to hold the results. An acceptable length is returned in NZ.

Description

The subroutine CCORL computes the discrete correlation of two complex sequences *x* and *y*. More precisely, let *n* be the length of *x* and *y*. If a circular correlation is desired, then IPAD must be set to zero (for *x* and *y* distinct) and two (for *x* = *y*). We set (on output)

$$n_z = n \qquad \text{if IPAD} = 0, 2$$

$$n_z = 2^\alpha 3^\beta 5^\gamma \geq 2n - 1 \qquad \text{if IPAD} = 1, 3$$

where α, β, γ are nonnegative integers yielding the smallest number of the type $2^\alpha 3^\beta 5^\gamma$ satisfying the inequality. Once n_z is determined, we pad out the vectors with zeroes. Then, we compute

$$z_i = \sum_{j=1}^{n_z} x_{i+j-1} \overline{y}_j$$

where the index on *x* is interpreted as a positive number between one and n_z, modulo n_z. Note that this means that

$$z_{n_z - k}$$

contains the correlation of $x(\cdot - \text{k} - 1)$ with *y* as $k = 0, 1, \ldots, n_z/2$. Thus, if $x(k-1) = y(k)$ for all *k*, then we would expect

$$\Re z_{n_z}$$

to be the largest component of $\Re z$.

The technique used to compute the z_i's is based on the fact that the (complex discrete) Fourier transform maps correlation into multiplication. Thus, the Fourier transform of *z* is given by

$$\hat{z}_j = \hat{x}_j \overline{\hat{y}_j}$$

where

$$\hat{z}_j = \sum_{m=1}^{n_z} z_m e^{-2\pi i(m-1)(j-1)/n_z}$$

Thus, the technique used here to compute the correlation is to take the discrete Fourier transform of x and the conjugate of the discrete Fourier transform of y, multiply the results together component-wise, and then take the inverse transform of this product. It is very important to make sure that n_z is a product of small primes if `IPAD` is set to zero or two. If n_z is a product of small primes, then the computational effort will be proportional to $n_z \log(n_z)$. If `IPAD` is one or three, then a good value is chosen for n_z so that the Fourier transforms are efficient and $n_z \geq 2n - 1$. This will mean that both vectors will be padded with zeroes.

INLAP

Computes the inverse Laplace transform of a complex function.

Required Arguments

F — User-supplied `FUNCTION` to which the inverse Laplace transform will be computed. The form is `F(Z)`, where

`Z` – Complex argument. (Input)
`F` – The complex function value. (Output)

`F` must be declared `EXTERNAL` in the calling program. `F` should also be declared `COMPLEX`.

T — Array of length `N` containing the points at which the inverse Laplace transform is desired. (Input)
`T(I)` must be greater than zero for all `I`.

FINV — Array of length `N` whose `I`-th component contains the approximate value of the Laplace transform at the point `T(I)`. (Output)

Optional Arguments

N — Number of points at which the inverse Laplace transform is desired. (Input)
Default: `N` = size (`T`,1).

ALPHA — An estimate for the maximum of the real parts of the singularities of `F`. If unknown, set `ALPHA` = 0. (Input)
Default: `ALPHA` = 0.0.

KMAX — The number of function evaluations allowed for each `T(I)`. (Input)
Default: `KMAX` = 500.

RELERR — The relative accuracy desired.   (Input)
Default: `RELERR` = 1.1920929e-5 for single precision and 2.22d-10 for double precision.

FORTRAN 90 Interface

Generic: `CALL INLAP (F, T, FINV [,…])`

Specific: The specific interface names are `S_INLAP` and `D_INLAP`.

FORTRAN 77 Interface

Single: `CALL INLAP (F, N, T, ALPHA, RELERR, KMAX, FINV)`

Double: The double precision name is `DINLAP`.

Example

We invert the Laplace transform of the simple function $(s-1)^{-2}$ and print the computed answer, the true solution and the difference at five different points. The correct inverse transform is xe^x.

```
      USE INLAP_INT
      USE UMACH_INT
      INTEGER    I, KMAX, N, NOUT
      REAL       ALPHA, DIF(5), EXP, FINV(5), FLOAT, RELERR, T(5), &
                TRUE(5)
      COMPLEX    F
      INTRINSIC  EXP, FLOAT
      EXTERNAL   F
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!
      DO 10  I=1, 5
         T(I) = FLOAT(I) - 0.5
   10 CONTINUE
      N      = 5
      ALPHA  = 1.0E0
      RELERR = 5.0E-4
      CALL INLAP (F, T, FINV, ALPHA=ALPHA, RELERR=RELERR)
!                                 Evaluate the true solution and the
!                                 difference
      DO 20  I=1, 5
         TRUE(I) = T(I)*EXP(T(I))
         DIF(I) = TRUE(I) - FINV(I)
   20 CONTINUE
!
      WRITE (NOUT,99999) (T(I),FINV(I),TRUE(I),DIF(I),I=1,5)
99999 FORMAT (7X, ’T’, 8X, ’FINV’, 9X, ’TRUE’, 9X, ’DIFF’, /, &
            5(1X,E9.1,3X,1PE10.3,3X,1PE10.3,3X,1PE10.3,/))
      END
!
      COMPLEX FUNCTION F (S)
      COMPLEX    S
      F = 1./(S-1.)**2
      RETURN
      END
```

Output

```
   T          FINV          TRUE          DIFF
0.5E+00    8.244E-01    8.244E-01   -4.768E-06
1.5E+00    6.723E+00    6.723E+00   -3.481E-05
2.5E+00    3.046E+01    3.046E+01   -1.678E-04
3.5E+00    1.159E+02    1.159E+02   -6.027E-04
4.5E+00    4.051E+02    4.051E+02   -2.106E-03
```

Comments

Informational errors

Type	Code	
4	1	The algorithm was not able to achieve the accuracy requested within KMAX function evaluations for some T(I).
4	2	Overflow is occurring for a particular value of T.

Description

The routine INLAP computes the inverse Laplace transform of a complex-valued function. Recall that if *f* is a function that vanishes on the negative real axis, then we can define the Laplace transform of *f* by

$$L[f](s) := \int_0^\infty e^{-sx} f(x)\,dx$$

It is assumed that for some value of *s* the integrand is absolutely integrable.

The computation of the inverse Laplace transform is based on applying the epsilon algorithm to the complex Fourier series obtained as a discrete approximation to the inversion integral. The initial algorithm was proposed by K.S. Crump (1976) but was significantly improved by de Hoog et al. (1982). Given a complex-valued transform $F(s) = L[f](s)$, the trapezoidal rule gives the approximation to the inverse transform

$$g(t) = \left(e^{\alpha t}/T\right)\Re\left\{\frac{1}{2}F(\alpha) + \sum_{k=1}^{\infty} F(\alpha + \frac{ik\pi}{T})\exp(\frac{ik\pi t}{T})\right\}$$

This is the real part of the sum of a complex power series in $z = \exp(i\pi t/T)$, and the algorithm accelerates the convergence of the partial sums of this power series by using the epsilon algorithm to compute the corresponding diagonal Pade approximants. The algorithm attempts to choose the order of the Pade approximant to obtain the specified relative accuracy while not exceeding the maximum number of function evaluations allowed. The parameter α is an estimate for the maximum of the real parts of the singularities of *F*, and an incorrect choice of α may give false convergence. Even in cases where the correct value of α is unknown, the algorithm will attempt to estimate an acceptable value. Assuming satisfactory convergence, the discretization error $E := g - f$ satisfies

$$E = \sum_{n=1}^{\infty} e^{-2n\alpha T} f\left(2nT + t\right)$$

It follows that if $|f(t)| \le Me^{\beta t}$, then we can estimate the expression above to obtain (for $0 \le t \le 2T$)

$$E \le Me^{\alpha t} / \left(e^{2T(\alpha - \beta)} - 1\right)$$

SINLP

Computes the inverse Laplace transform of a complex function.

Required Arguments

F — User-supplied `FUNCTION` to which the inverse Laplace transform will be computed. The form is `F`(`Z`), where

`Z` — Complex argument. (Input)
`F` — The complex function value. (Output)

`F` must be declared `EXTERNAL` in the calling program. `F` must also be declared `COMPLEX`.

T — Vector of length `N` containing points at which the inverse Laplace transform is desired. (Input)
`T`(`I`) must be greater than zero for all `I`.

FINV — Vector of length `N` whose `I`-th component contains the approximate value of the inverse Laplace transform at the point `T`(`I`). (Output)

Optional Arguments

N — The number of points at which the inverse Laplace transform is desired. (Input)
Default: `N` = size (`T`,1).

SIGMA0 — An estimate for the maximum of the real parts of the singularities of `F`. (Input)
If unknown, set `SIGMA0` = 0.0.
Default: `SIGMA0` = 0.e0.

EPSTOL — The required absolute uniform pseudo accuracy for the coefficients and inverse Laplace transform values. (Input)
Default: `EPSTOL` = 1.1920929e-5 for single precision and 2.22d-10 for double precision.

ERRVEC — Vector of length eight containing diagnostic information. (Output)
All components depend on the intermediately generated Laguerre coefficients. See Comments.

FORTRAN 90 Interface

Generic: `CALL SINLP (F, T, FINV [,…])`

Specific: The specific interface names are `S_SINLP` and `D_SINLP`.

FORTRAN 77 Interface

Single: `CALL SINLP (F, N, T, SIGMA0, EPSTOL, ERRVEC, FINV)`

Double: The double precision name is `DSINLP`.

Example

We invert the Laplace transform of the simple function $(s - 1)^{-2}$ and print the computed answer, the true solution, and the difference at five different points. The correct inverse transform is xe^x.

```
      USE SINLP_INT
      USE UMACH_INT
      INTEGER    I, NOUT
      REAL       DIF(5), ERRVEC(8), EXP, FINV(5), FLOAT, RELERR, &
                SIGMA0, T(5), TRUE(5)
      COMPLEX    F
      INTRINSIC  EXP, FLOAT
      EXTERNAL   F
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!
      DO 10  I=1, 5
         T(I) = FLOAT(I) - 0.5
   10 CONTINUE
      SIGMA0 = 1.0E0
      RELERR = 5.0E-4
      EPSTOL = 1.0E-4
      CALL SINLP (F, T, FINV, SIGMA0=SIGMA0, EPSTOL=RELERR)
!                                 Evaluate the true solution and the
!                                 difference
      DO 20  I=1, 5
         TRUE(I) = T(I)*EXP(T(I))
         DIF(I) = TRUE(I) - FINV(I)
   20 CONTINUE
!
      WRITE (NOUT,99999) (T(I),FINV(I),TRUE(I),DIF(I),I=1,5)
99999 FORMAT (7X, ’T’, 8X, ’FINV’, 9X, ’TRUE’, 9X, ’DIFF’, /, &
            5(1X,E9.1,3X,1PE10.3,3X,1PE10.3,3X,1PE10.3,/))
      END
!
      COMPLEX FUNCTION F (S)
      COMPLEX    S
!
      F = 1./(S-1.)**2
      RETURN
      END
```

Output

```
    T        FINV          TRUE          DIFF
0.5E+00    8.244E-01    8.244E-01   -2.086E-06
1.5E+00    6.723E+00    6.723E+00   -8.583E-06
2.5E+00    3.046E+01    3.046E+01    0.000E+00
3.5E+00    1.159E+02    1.159E+02    2.289E-05
4.5E+00    4.051E+02    4.051E+02   -2.136E-04
```

Comments

1. Workspace may be explicitly provided, if desired, by use of S2NLP/DS2NLP. The reference is:

   ```
   CALL S2NLP (F, N, T, SIGMA0, EPSTOL, ERRVEC, FINV,
   SIGMA, BVALUE, MTOP, WK, IFLOVC)
   ```

   The additional arguments are as follows:

   SIGMA — The first parameter of the Laguerre expansion. If SIGMA is not greater than SIGMA0, it is reset to SIGMA0 + 0.7.   (Input)

   BVALUE — The second parameter of the Laguerre expansion. If BVALUE is less than 2.0 * (SIGMA – SIGMA0), it is reset to 2.5 * (SIGMA – SIGMA0).   (Input)

   MTOP — An upper limit on the number of coefficients to be computed in the Laguerre expansion. MTOP must be a multiple of four. Note that the maximum number of Laplace transform evaluations is MTOP/2 + 2. (Default: 1024.)   (Input)

   WK — Real work vector of length 9 * MTOP/4.

   IFLOVC — Integer vector of length N, the I-th component of which contains the overflow/underflow indicator for the computed value of FINV(I).   (Output) See Comment 3.

2. Informational errors

Type	Code	
1	1	Normal termination, but with estimated error bounds slightly larger than EPSTOL. Note, however, that the actual errors on the final results may be smaller than EPSTOL as bounds independent of T are pessimistic.
3	2	Normal calculation, terminated early at the roundoff error level estimate because this estimate exceeds the required accuracy (usually due to overly optimistic expectation by the user about attainable accuracy).

4	3	The decay rate of the coefficients is too small. It may improve results to use S2NLP and increase MTOP.
4	4	The decay rate of the coefficients is too small. In addition, the roundoff error level is such that required accuracy cannot be reached.
4	5	No error bounds are returned as the behavior of the coefficients does not enable reasonable prediction. Results are probably wrong. Check the value of SIGMA0. In this case, each of ERRVEC(J), J = 1, …, 5, is set to – 1.0.

3. The following are descriptions of the vectors ERRVEC and IFLOVC.

ERRVEC — Real vector of length eight.

ERRVEC(1) = Overall estimate of the pseudo error, ERRVEC(2) + ERRVEC(3) + ERRVEC(4). Pseudo error = absolute error / exp(sigma * tvalue).

ERRVEC(2) = Estimate of the pseudo discretization error.

ERRVEC(3) = Estimate of the pseudo truncation error.

ERRVEC(4) = Estimate of the pseudo condition error on the basis of minimal noise levels in the function values.

ERRVEC(5) = K, the coefficient of the decay function for ACOEF, the coefficients of the Laguerre expansion.

ERRVEC(6) = R, the base of the decay function for ACOEF. Here abs(ACOEF (J + 1)).LE.K/R**J for J.GE.MACT/2, where MACT is the number of Laguerre coefficients actually computed.

ERRVEC(7) = ALPHA, the logarithm of the largest ACOEF.

ERRVEC(8) = BETA, the logarithm of the smallest nonzero ACOEF.

IFLOVC — Integer vector of length N containing the overflow/underflow indicators for FINV. For each I, the value of IFLOVC(I) signifies the following.

0 = Normal termination.

1 = The value of the inverse Laplace transform is found to be too large to be representable; FINV(I) is set to AMACH(6).

−1 = The value of the inverse Laplace transform is found to be too small to be representable; FINV(I) is set to 0.0.

2 = The value of the inverse Laplace transform is estimated to be too large, even before the series expansion, to be representable; FINV(I) is set to AMACH(6).

−2 = The value of the inverse Laplace transform is estimated to be too small, even before the series expansion, to be representable; FINV(I) is set to 0.0.

Description

The routine SINLP computes the inverse Laplace transform of a complex-valued function. Recall that if *f* is a function that vanishes on the negative real axis, then we can define the Laplace transform of *f* by

$$L[f](s) := \int_0^\infty e^{-sx} f(x)\,dx$$

It is assumed that for some value of *s* the integrand is absolutely integrable.

The computation of the inverse Laplace transform is based on a modification of Weeks' method (see W.T. Weeks (1966)) due to B.S. Garbow et. al. (1988). This method is suitable when *f* has continuous derivatives of all orders on [0, ∞). In this situation, this routine should be used in place of the IMSL routine INLAP (page 1078). It is especially efficient when multiple function values are desired. In particular, given a complex-valued function $F(s) = L[f](s)$, we can expand *f* in a Laguerre series whose coefficients are determined by *F*. This is fully described in B.S. Garbow et. al. (1988) and Lyness and Giunta (1986).

The algorithm attempts to return approximations $g(t)$ to $f(t)$ satisfying

$$\left|\frac{g(t)-f(t)}{e^{\sigma t}}\right| < \varepsilon$$

where ε := EPSTOL and σ := SIGMA > SIGMA0. The expression on the left is called the pseudo error. An estimate of the pseudo error is available in ERRVEC(1).

The first step in the method is to transform *F* to ϕ where

$$\phi(z) = \frac{b}{1-z} F\left(\frac{b}{1-z} - \frac{b}{2} + \sigma\right)$$

Then, if *f* is smooth, it is known that ϕ is analytic in the unit disc of the complex plane and hence has a Taylor series expansion

$$\phi(z) = \sum_{s=0}^{\infty} a_s z^s$$

which converges for all *z* whose absolute value is less than the radius of convergence R_c. This number is estimated in ERRVEC(6). In ERRVEC(5), we estimate the smallest number *K* which satisfies

$$|a_s| < \frac{K}{R^s}$$

for all $R < R_c$.

The coefficients of the Taylor series for ϕ can be used to expand *f* in a Laguerre series

$$f(t) = e^{\sigma t} \sum_{s=0}^{\infty} a_s e^{-bt/2} L_s(bt)$$

Chapter 7: Nonlinear Equations

Routines

Usage Notes

Zeros of a Polynomial

A polynomial function of degree n can be expressed as follows:

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0$$

where $a_n \neq 0$.

There are three routines for zeros of a polynomial. The routines `ZPLRC` (page 1148) and `ZPORC` (page 1150) find zeros of the polynomial with real coefficients while the routine `ZPOCC` (page 1152) finds zeros of the polynomial with complex coefficients.

The Jenkins-Traub method is used for the routines `ZPORC` and `ZPOCC`; whereas `ZPLRC` uses the Laguerre method. Both methods perform well in comparison with other methods. The Jenkins-Traub algorithm usually runs faster than the Laguerre method. Furthermore, the routine `ZANLY` (page 1153) in the next section can also be used for the complex polynomial.

Zero(s) of a Function

The routines ZANLY (page 1153) and ZREAL (page 1159) use Müller's method to find the zeros of a complex analytic function and real zeros of a real function, respectively. The routine ZBREN (page 1156) finds a zero of a real function, using an algorithm that is a combination of interpolation and bisection. This algorithm requires the user to supply two points such that the function values at these two points have opposite sign. For functions where it is difficult to obtain two such points, ZREAL can be used.

Root of System of Equations

A system of equations can be stated as follows:

$$f_i(x) = 0, \text{ for } i = 1, 2, \ldots, n$$

where $x \varepsilon \mathbf{R}^n$.

The routines NEQNF (page 1162) and NEQNJ (page 1165) use a modified Powell hybrid method to find a zero of a system of nonlinear equations. The difference between these two routines is that the Jacobian is estimated by a finite-difference method in NEQNF, whereas the user has to provide the Jacobian for NEQNJ. It is advised that the Jacobian-checking routine, CHJAC (page 952), be used to ensure the accuracy of the user-supplied Jacobian.

The routines NEQBF (page 1169) and NEQBJ (page 1174) use a secant method with Broyden's update to find a zero of a system of nonlinear equations. The difference between these two routines is that the Jacobian is estimated by a finite-difference method in NEQBF; whereas the user has to provide the Jacobian for NEQBJ. For more details, see Dennis and Schnabel (1983, Chapter 8).

ZPLRC

Finds the zeros of a polynomial with real coefficients using Laguerre's method.

Required Arguments

COEFF — Vector of length NDEG + 1 containing the coefficients of the polynomial in increasing order by degree. (Input)
The polynomial is COEFF(NDEG + 1) * Z**NDEG + COEFF(NDEG) * Z**(NDEG – 1) + … + COEFF(1).

ROOT — Complex vector of length NDEG containing the zeros of the polynomial. (Output)

Optional Arguments

NDEG — Degree of the polynomial. 1 ≤ NDEG ≤ 100 (Input)
Default: NDEG = size (COEFF,1) – 1.

FORTRAN 90 Interface

Generic: `CALL ZPLRC (COEFF, ROOT [,…])`

Specific: The specific interface names are `S_ZPLRC` and `D_ZPLRC`.

FORTRAN 77 Interface

Single: `CALL ZPLRC (NDEG, COEFF, ROOT)`

Double: The double precision name is `DZPLRC`.

Example

This example finds the zeros of the third-degree polynomial

$$p(z) = z^3 - 3z^2 + 4z - 2$$

where z is a complex variable.

```
      USE ZPLRC_INT
      USE WRCRN_INT
!                                 Declare variables
      INTEGER    NDEG
      PARAMETER  (NDEG=3)
!
      REAL       COEFF(NDEG+1)
      COMPLEX    ZERO(NDEG)
!                                 Set values of COEFF
!                                 COEFF = (-2.0  4.0 -3.0  1.0)
!
      DATA COEFF/-2.0, 4.0, -3.0, 1.0/
!
      CALL ZPLRC (COEFF, ZERO, NDEG)
!
      CALL WRCRN ('The zeros found are', ZERO, 1, NDEG, 1)
!
      END
```

Output

```
              The zeros found are
             1                 2                 3
( 1.000, 1.000)  ( 1.000,-1.000)  ( 1.000, 0.000)
```

Comments

Informational errors

Type	Code	
3	1	The first several coefficients of the polynomial are equal to zero. Several of the last roots will be set to machine infinity to compensate for this problem.

3	2	Fewer than `NDEG` zeros were found. The `ROOT` vector will contain the value for machine infinity in the locations that do not contain zeros.

Description

Routine `ZPLRC` computes the n zeros of the polynomial

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0$$

where the coefficients a_i for $i = 0, 1, \ldots, n$ are real and n is the degree of the polynomial.

The routine `ZPLRC` is a modification of B.T. Smith's routine `ZERPOL` (Smith 1967) that uses Laguerre's method. Laguerre's method is cubically convergent for isolated zeros and linearly convergent for multiple zeros. The maximum length of the step between successive iterates is restricted so that each new iterate lies inside a region about the previous iterate known to contain a zero of the polynomial. An iterate is accepted as a zero when the polynomial value at that iterate is smaller than a computed bound for the rounding error in the polynomial value at that iterate. The original polynomial is deflated after each real zero or pair of complex zeros is found. Subsequent zeros are found using the deflated polynomial.

ZPORC

Finds the zeros of a polynomial with real coefficients using the Jenkins-Traub three-stage algorithm.

Required Arguments

COEFF — Vector of length `NDEG` + 1 containing the coefficients of the polynomial in increasing order by degree. (Input)
The polynomial is `COEFF`(`NDEG` + 1)`*Z**NDEG` + `COEFF`(`NDEG`) `* Z**`(`NDEG` –1) + … + `COEFF`(1).

ROOT — Complex vector of length `NDEG` containing the zeros of the polynomial. (Output)

Optional Arguments

NDEG — Degree of the polynomial. 1 ≤ `NDEG` ≤ 100 (Input)
Default: `NDEG` = size (`COEFF`,1) – 1.

FORTRAN 90 Interface

Generic: `CALL ZPORC (COEFF, ROOT [,…])`

Specific: The specific interface names are `S_ZPORC` and `D_ZPORC`.

FORTRAN 77 Interface

Single: `CALL ZPORC (NDEG, COEFF, ROOT)`

Double: The double precision name is DZPORC.

Example

This example finds the zeros of the third-degree polynomial

$$p(z) = z^3 - 3z^2 + 4z - 2$$

where z is a complex variable.

```
      USE ZPORC_INT
      USE WRCRN_INT
!                                 Declare variables
      INTEGER    NDEG
      PARAMETER  (NDEG=3)
!
      REAL       COEFF(NDEG+1)
      COMPLEX    ZERO(NDEG)
!                                 Set values of COEFF
!                                 COEFF = (-2.0  4.0 -3.0  1.0)
!
      DATA COEFF/-2.0, 4.0, -3.0, 1.0/
!
      CALL ZPORC (COEFF, ZERO)
!
      CALL WRCRN ('The zeros found are', ZERO, 1, NDEG, 1)
!
      END
```

Output

```
              The zeros found are
             1                2                3
( 1.000, 0.000)  ( 1.000, 1.000)  ( 1.000,-1.000)
```

Comments

Informational errors

Type	Code	
3	1	The first several coefficients of the polynomial are equal to zero. Several of the last roots will be set to machine infinity to compensate for this problem.
3	2	Fewer than NDEG zeros were found. The ROOT vector will contain the value for machine infinity in the locations that do not contain zeros.

Description

Routine ZPORC computes the n zeros of the polynomial

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0$$

where the coefficients a_i for $i = 0, 1, \ldots, n$ are real and n is the degree of the polynomial.

The routine ZPORC uses the Jenkins-Traub three-stage algorithm (Jenkins and Traub 1970; Jenkins 1975). The zeros are computed one at a time for real zeros or two at a time for complex conjugate pairs. As the zeros are found, the real zero or quadratic factor is removed by polynomial deflation.

ZPOCC

Finds the zeros of a polynomial with complex coefficients.

Required Arguments

COEFF — Complex vector of length NDEG + 1 containing the coefficients of the polynomial in increasing order by degree. (Input)
The polynomial is COEFF(NDEG + 1) * Z**NDEG + COEFF(NDEG) * Z**(NDEG – 1) + … + COEFF(1).

ROOT — Complex vector of length NDEG containing the zeros of the polynomial. (Output)

Optional Arguments

NDEG – Degree of the polynomial. 1 ≤ NDEG < 50 (Input)
Default: NDEG = size (COEFF,1) – 1.

FORTRAN 90 Interface

Generic: CALL ZPOCC (COEFF, ROOT [,…])

Specific: The specific interface names are S_ZPOCC and D_ZPOCC.

FORTRAN 77 Interface

Single: CALL ZPOCC (NDEG, COEFF, ROOT)

Double: The double precision name is DZPOCC.

Example

This example finds the zeros of the third-degree polynomial

$$p(z) = z^3 - (3 + 6i)z^2 - (8 - 12i)z + 10$$

where z is a complex variable.

```
      USE ZPOCC_INT
      USE WRCRN_INT
!                                  Declare variables
      INTEGER    NDEG
      PARAMETER  (NDEG=3)
!
      COMPLEX    COEFF(NDEG+1), ZERO(NDEG)
!                                  Set values of COEFF
!                                  COEFF = ( 10.0 +  0.0i )
!                                          ( -8.0 + 12.0i )
!                                          ( -3.0 -  6.0i )
!                                          (  1.0 +  0.0i )
!
      DATA COEFF/(10.0,0.0), (-8.0,12.0), (-3.0,-6.0), (1.0,0.0)/
!
      CALL ZPOCC (COEFF, ZERO)
!
      CALL WRCRN ('The zeros found are', ZERO, 1, NDEG, 1)
!
      END
```

Output

```
              The zeros found are
             1                 2                 3
( 1.000, 1.000)  ( 1.000, 2.000)  ( 1.000, 3.000)
```

Comments

Informational errors

Type	Code	
3	1	The first several coefficients of the polynomial are equal to zero. Several of the last roots will be set to machine infinity to compensate for this problem.
3	2	Fewer than NDEG zeros were found. The ROOT vector will contain the value for machine infinity in the locations that do not contain zeros.

Description

Routine ZPOCC computes the *n* zeros of the polynomial

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0$$

where the coefficients a_i for $i = 0, 1, \ldots, n$ are real and n is the degree of the polynomial.

The routine ZPOCC uses the Jenkins-Traub three-stage complex algorithm (Jenkins and Traub 1970, 1972). The zeros are computed one at a time in roughly increasing order of modulus. As each zero is found, the polynomial is deflated to one of lower degree.

ZANLY

Finds the zeros of a univariate complex function using Müller's method.

Required Arguments

F — User-supplied COMPLEX FUNCTION to compute the value of the function of which the zeros will be found. The form is F(Z), where

Z — The complex value at which the function is evaluated. (Input)
Z should not be changed by F.

F — The computed complex function value at the point Z. (Output)
F must be declared EXTERNAL in the calling program.

Z — A complex vector of length NKNOWN + NNEW. (Output)
Z(1), …, Z(NKNOWN) contain the known zeros. Z(NKNOWN + 1), …, Z(NKNOWN + NNEW) contain the new zeros found by ZANLY. If ZINIT is not needed, ZINIT and Z can share the same storage locations.

Optional Arguments

ERRABS — First stopping criterion. (Input)
Let FP(Z) = F(Z)/P where P = (Z – Z(1)) * (Z – Z(2)) *…* (Z – Z(K – 1)) and Z(1), …, Z(K – 1) are previously found zeros. If (CABS(F(Z)).LE.ERRABS.AND.CABS(FP(Z)).LE.ERRABS), then Z is accepted as a zero.
Default: ERRABS = 1.e-4 for single precision and 1.d-8 for double precision.

ERRREL — Second stopping criterion is the relative error. (Input)
A zero is accepted if the difference in two successive approximations to this zero is within ERRREL. ERRREL must be less than 0.01; otherwise, 0.01 will be used.
Default: ERRREL = 1.e-4 for single precision and 1.d-8 for double precision.

NKNOWN — The number of previously known zeros, if any, that must be stored in ZINIT(1), …, ZINIT(NKNOWN) prior to entry to ZANLY. (Input)
NKNOWN must be set equal to zero if no zeros are known.
Default: NKNOWN = 0.

NNEW — The number of new zeros to be found by ZANLY. (Input)
Default: NNEW = 1.

NGUESS — The number of initial guesses provided. (Input)
These guesses must be stored in ZINIT(NKNOWN + 1), …, ZINIT(NKNOWN + NGUESS). NGUESS must be set equal to zero if no guesses are provided.
Default: NGUESS = 0.

ITMAX — The maximum allowable number of iterations per zero. (Input)
Default: ITMAX = 100.

ZINIT — A complex vector of length NKNOWN + NNEW. (Input)
ZINIT(1), …, ZINIT(NKNOWN) must contain the known zeros. ZINIT(NKNOWN + 1), …, ZINIT(NKNOWN + NNEW) may, on user option, contain initial guesses for the NNEW new zeros that are to be computed. If the user does not provide an initial guess, zero is used.

INFO — An integer vector of length NKNOWN + NNEW. (Output)
INFO(J) contains the number of iterations used in finding the J-th zero when convergence was achieved. If convergence was not obtained in ITMAX iterations, INFO(J) will be greater than ITMAX.

FORTRAN 90 Interface

Generic: `CALL ZANLY (F, Z [,…])`

Specific: The specific interface names are S_ZANLY and D_ZANLY.

FORTRAN 77 Interface

Single:
```
CALL ZANLY (F, ERRABS, ERRREL, NKNOWN, NNEW, NGUESS,
ZINIT, ITMAX, Z, INFO)
```

Double: The double precision name is DZANLY.

Comments

1. Informational error

Type	Code	
3	1	Failure to converge within ITMAX iterations for at least one of the NNEW new roots.

2. Routine ZANLY always returns the last approximation for zero J in Z(J). If the convergence criterion is satisfied, then INFO(J) is less than or equal to ITMAX. If the convergence criterion is not satisfied, then INFO(J) is set to either ITMAX + 1 or ITMAX + K, with K greater than 1. INFO(J) = ITMAX + 1 indicates that ZANLY did not obtain convergence in the allowed number of iterations. In this case, the user may wish to set ITMAX to a larger value. INFO(J) = ITMAX + K means that convergence was obtained (on iteration K) for the deflated function FP(Z) = F(Z)/((Z − Z(1)) … (Z − Z(J − 1))) but failed for F(Z). In this case, better initial guesses might help or it might be necessary to relax the convergence criterion.

Description

Müller’s method with deflation is used. It assumes that the complex function $f(z)$ has at least two continuous derivatives. For more details, see Müller (1965).

Example

This example finds the zeros of the equation $f(z) = z^3 + 5z^2 + 9z + 45$, where z is a complex variable.

```
      USE ZANLY_INT
      USE WRCRN_INT
!                                 Declare variables
      INTEGER    INFO(3), NGUESS, NNEW
      COMPLEX    F, Z(3), ZINIT(3)
      EXTERNAL   F
!                                 Set the guessed zero values in ZINIT
!
!                                 ZINIT = (1.0+1.0i 1.0+1.0i 1.0+1.0i)
      DATA ZINIT/3*(1.0,1.0)/
!                                 Set values for all input parameters
      NNEW   = 3
      NGUESS = 3
!                                 Find the zeros of F
      CALL ZANLY (F, Z, NNEW=NNEW, NGUESS=NGUESS, &
                 ZINIT=ZINIT, INFO=INFO)
!                                 Print results
      CALL WRCRN ('The zeros are', Z)
      END
!                                 External complex function
      COMPLEX FUNCTION F (Z)
      COMPLEX    Z
!
      F = Z**3 + 5.0*Z**2 + 9.0*Z + 45.0
      RETURN
      END
```

Output

```
              The zeros are
             1                2                3
( 0.000, 3.000)  ( 0.000,-3.000)  (-5.000, 0.000)
```

ZBREN

Finds a zero of a real function that changes sign in a given interval.

Required Arguments

F — User-supplied `FUNCTION` to compute the value of the function of which a zero will be found. The form is `F`(`X`), where

`X` — The point at which the function is evaluated. (Input)
`X` should not be changed by `F`.

`F` — The computed function value at the point `X`. (Output)
`F` must be declared `EXTERNAL` in the calling program.

A — See B. (Input/Output)

B — On input, the user must supply two points, A and B, such that F(A) and F(B) are opposite in sign. (Input/Output)
On output, both A and B are altered. B will contain the best approximation to the zero of F.

Optional Arguments

ERRABS — First stopping criterion. (Input)
A zero, B, is accepted if ABS(F(B)) is less than or equal to ERRABS. ERRABS may be set to zero.
Default: ERRABS = 1.e-4 for single precision and 1.d-8 for double precision.

ERRREL — Second stopping criterion is the relative error. (Input)
A zero is accepted if the change between two successive approximations to this zero is within ERRREL.
Default: ERRREL = 1.e-4 for single precision and 1.d-8 for double precision.

MAXFN — On input, MAXFN specifies an upper bound on the number of function evaluations required for convergence. (Input/Output)
On output, MAXFN will contain the actual number of function evaluations used.
Default: MAXFN = 100.

FORTRAN 90 Interface

Generic: CALL ZBREN (F, A, B [,…])

Specific: The specific interface names are S_ZBREN and D_ZBREN.

FORTRAN 77 Interface

Single: CALL ZBREN (F, ERRABS, ERRREL, A, B, MAXFN)

Double: The double precision name is DZBREN.

Example

This example finds a zero of the function

$$f(x) = x^2 + x - 2$$

in the interval (− 10.0, 0.0).

```
      USE ZBREN_INT
      USE UMACH_INT
!                                 Declare variables
      REAL       ERRABS, ERRREL
!
      INTEGER    NOUT
      REAL       A, B, F
      EXTERNAL   F
!                                 Set values of A, B, ERRABS,
!                                 ERRREL, MAXFN
      A      = -10.0
      B      = 0.0
      ERRABS = 0.0
      ERRREL = 0.001
      MAXFN  = 100
!
      CALL UMACH (2, NOUT)
!                                 Find zero of F
      CALL ZBREN (F, A, B, ERRABS=ERRABS, ERRREL=ERRREL, MAXFN=MAXFN)
!
      WRITE (NOUT,99999) B, MAXFN
99999 FORMAT ('  The best approximation to the zero of F is equal to', &
            F5.1, '.', /, '  The number of function evaluations', &
            ' required was ', I2, '.', //)
!
      END
!
      REAL FUNCTION F (X)
      REAL       X
!
      F = X**2 + X - 2.0
      RETURN
      END
```

Output

```
The best approximation to the zero of F is equal to -2.0.
The number of function evaluations required was 12.
```

Comments

1. Informational error

Type	Code	
4	1	Failure to converge in MAXFN function evaluations.

2. On exit from ZBREN without any error message, A and B satisfy the following:

   $F(A)F(B) \le 0.0$
   $|F(B)| \le |F(A)|$, and
   either $|F(B)| \le$ ERRABS or
   $|A - B| \le \max(|B|, 0.1) *$ ERRREL.

   The presence of 0.1 in the stopping criterion causes leading zeros to the right of the decimal point to be counted as significant digits. Scaling may be required in order to accurately determine a zero of small magnitude.

3. ZBREN is guaranteed to convergence within K function evaluations, where $K = (\ln((B - A)/D) + 1.0)^2$, and

$$\left(\mathrm{D} = \min_{x \in (\mathrm{A},\mathrm{B})} \left(\max\left(|\mathrm{x}|, 0.1\right) * \mathrm{ERRREL}\right)\right)$$

This is an upper bound on the number of evaluations. Rarely does the actual number of evaluations used by ZBREN exceed

$$\sqrt{\mathrm{K}}$$

```
D can be computed as follows:
P = AMAX1(0.1, AMIN1(|A|, |B|))
IF((A – 0.1) * (B – 0.1) < 0.0) P = 0.1,
D = P * ERRREL
```

Description

The algorithm used by ZBREN is a combination of linear interpolation, inverse quadratic interpolation, and bisection. Convergence is usually superlinear and is never much slower than the rate for the bisection method. See Brent (1971) for a more detailed account of this algorithm.

ZREAL

Finds the real zeros of a real function using Müller's method.

Required Arguments

F — User-supplied FUNCTION to compute the value of the function of which a zero will be found. The form is F(X), where

X – The point at which the function is evaluated. (Input)
X should not be changed by F.

F – The computed function value at the point X. (Output)
F must be declared EXTERNAL in the calling program.

X — A vector of length NROOT. (Output)
X contains the computed zeros.

Optional Arguments

ERRABS — First stopping criterion. (Input)
A zero X(I) is accepted if ABS(F(X(I)).LT. ERRABS.
Default: ERRABS = 1.e-4 for single precision and 1.d-8 for double precision.

ERRREL — Second stopping criterion is the relative error. (Input)
A zero X(I) is accepted if the relative change of two successive approximations to X(I) is less than ERRREL.
Default: ERRREL = 1.e-4 for single precision and 1.d-8 for double precision.

EPS — See ETA.   (Input)
Default: EPS = 1.e-4 for single precision and 1.d-8 for double precision.

ETA — Spread criteria for multiple zeros.   (Input)
If the zero X(I) has been computed and ABS(X(I) − X(J)).LT.EPS, where X(J) is a previously computed zero, then the computation is restarted with a guess equal to X(I) + ETA.
Default: ETA = .01.

NROOT — The number of zeros to be found by ZREAL.   (Input)
Default: NROOT = 1.

ITMAX — The maximum allowable number of iterations per zero.   (Input)
Default: ITMAX = 100.

XGUESS — A vector of length NROOT.   (Input)
XGUESS contains the initial guesses for the zeros.
Default: XGUESS = 0.0.

INFO — An integer vector of length NROOT.   (Output)
INFO(J) contains the number of iterations used in finding the J-th zero when convergence was achieved. If convergence was not obtained in ITMAX iterations, INFO(J) will be greater than ITMAX.

FORTRAN 90 Interface

Generic: CALL ZREAL (F, X [,…])

Specific: The specific interface names are S_ZREAL and D_ZREAL.

FORTRAN 77 Interface

Single: CALL ZREAL (F, ERRABS, ERRREL, EPS, ETA, NROOT, ITMAX, XGUESS, X, INFO)

Double: The double precision name is DZREAL.

Example

This example finds the real zeros of the second-degree polynomial

$$f(x) = x^2 + 2x - 6$$

with the initial guess (4.6, −193.3).

```
      USE ZREAL_INT
      USE WRRRN_INT
!                                 Declare variables
      INTEGER    NROOT
      REAL       EPS, ERRABS, ERRREL
      PARAMETER  (NROOT=2)
!
      INTEGER    INFO(NROOT)
      REAL       F, X(NROOT), XGUESS(NROOT)
      EXTERNAL   F
!                                 Set values of initial guess
!                                 XGUESS = (  4.6 -193.3)
!
      DATA XGUESS/4.6, -193.3/
!
      EPS    = 1.0E-5
      ERRABS = 1.0E-5
      ERRREL = 1.0E-5

!                                 Find the zeros
      CALL ZREAL (F, X, ERRABS=ERRABS, ERRREL=ERRREL, EPS=EPS, &
                 NROOT=NROOT, XGUESS=XGUESS)
!
      CALL WRRRN (’The zeros are’, X, 1, NROOT, 1)
!
      END
!
      REAL FUNCTION F (X)
      REAL       X
!
      F = X*X + 2.0*X - 6.0
      RETURN
      END
```

Output

```
The zeros are
    1       2
1.646  -3.646
```

Comments

1. Informational error

Type	Code	
3	1	Failure to converge within ITMAX iterations for at least one of the NROOT roots.

2. Routine ZREAL always returns the last approximation for zero J in X(J). If the convergence criterion is satisfied, then INFO(J) is less than or equal to ITMAX. If the convergence criterion is not satisfied, then INFO(J) is set to ITMAX + 1.

3. The routine ZREAL assumes that there exist NROOT distinct real zeros for the function F and that they can be reached from the initial guesses supplied. The routine is designed so that convergence to any single zero cannot be obtained from two different initial guesses.

4. Scaling the X vector in the function F may be required, if any of the zeros are known to be less than one.

Description

Routine ZREAL computes n real zeros of a real function f. Given a user-supplied function $f(x)$ and an n-vector of initial guesses $x_1, x_2, \ldots, x_n$, the routine uses Müller's method to locate n real zeros of f, that is, n real values of x for which $f(x) = 0$. The routine has two convergence criteria: the first requires that

$$\left| f\left(x_i^m\right) \right|$$

be less than ERRABS; the second requires that the relative change of any two successive approximations to an x_i be less than ERRREL. Here,

$$x_i^m$$

is the m-th approximation to x_i. Let ERRABS be ε_1, and ERRREL be ε_2.The criteria may be stated mathematically as follows:

Criterion 1:

$$\left| f\left(x_i^m\right) \right| < \varepsilon_1$$

Criterion 2:

$$\left| \frac{x_i^{m+1} - x_i^m}{x_i^m} \right| < \varepsilon_2$$

"Convergence" is the satisfaction of either criterion.

NEQNF

Solves a system of nonlinear equations using a modified Powell hybrid algorithm and a finite-difference approximation to the Jacobian.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the system of equations to be solved. The usage is CALL FCN (X, F, N), where

X – The point at which the functions are evaluated. (Input)
X should not be changed by FCN.

F – The computed function values at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

N — Length of X and F. (Input)

X — A vector of length N. (Output)
X contains the best estimate of the root found by NEQNF.

Optional Arguments

ERRREL — Stopping criterion. (Input)
The root is accepted if the relative error between two successive approximations to this root is less than ERRREL.
Default: ERRREL = 1.e-4 for single precision and 1.d-8 for double precision.

N – The number of equations to be solved and the number of unknowns. (Input)
Default: N = size (X,1).

ITMAX — The maximum allowable number of iterations. (Input)
The maximum number of calls to FCN is ITMAX * (N + 1). Suggested value ITMAX = 200.
Default: ITMAX = 200.

XGUESS — A vector of length N. (Input)
XGUESS contains the initial estimate of the root.
Default: XGUESS = 0.0.

FNORM — A scalar that has the value $F(1)^2 + \ldots + F(N)^2$ at the point X. (Output)

FORTRAN 90 Interface

Generic: `CALL NEQNF (FCN, X [,…])`

Specific: The specific interface names are S_NEQNF and D_NEQNF.

FORTRAN 77 Interface

Single: `CALL NEQNF (FCN, ERRREL, N, ITMAX, XGUESS, X, FNORM)`

Double: The double precision name is DNEQNF.

Example

The following 3 × 3 system of nonlinear equations

$$f_1(x) = x_1 + e^{x_1 - 1} + (x_2 + x_3)^2 - 27 = 0$$
$$f_2(x) = e^{x_2 - 2} / x_1 + x_3^2 - 10 = 0$$
$$f_3(x) = x_3 + \sin(x_2 - 2) + x_2^2 - 7 = 0$$

is solved with the initial guess (4.0, 4.0, 4.0).

```
      USE NEQNF_INT
      USE UMACH_INT
!                                  Declare variables
      INTEGER    N
      PARAMETER  (N=3)
!
      INTEGER    K, NOUT
      REAL       FNORM, X(N), XGUESS(N)
      EXTERNAL   FCN
!                                  Set values of initial guess
!                                  XGUESS = (  4.0  4.0  4.0 )
!
      DATA XGUESS/4.0, 4.0, 4.0/
!
!
      CALL UMACH (2, NOUT)
!                                  Find the solution
      CALL NEQNF (FCN, X, XGUESS=XGUESS, FNORM=FNORM)
!                                  Output
      WRITE (NOUT,99999) (X(K),K=1,N), FNORM
99999 FORMAT ('  The solution to the system is', /, '  X = (', 3F5.1, &
            ')', /, '  with FNORM =', F5.4, //)
!
      END
!                                  User-defined subroutine
      SUBROUTINE FCN (X, F, N)
      INTEGER    N
      REAL       X(N), F(N)
!
      REAL       EXP, SIN
      INTRINSIC  EXP, SIN
!
      F(1) = X(1) + EXP(X(1)-1.0) + (X(2)+X(3))*(X(2)+X(3)) - 27.0
      F(2) = EXP(X(2)-2.0)/X(1) + X(3)*X(3) - 10.0
      F(3) = X(3) + SIN(X(2)-2.0) + X(2)*X(2) - 7.0
      RETURN
      END
```

Output

```
The solution to the system is
X = (  1.0  2.0  3.0)
with FNORM =.0000
```

Comments

1. Workspace may be explicitly provided, if desired, by use of N2QNF/DN2QNF. The reference is:

   ```
   CALL N2QNF (FCN, ERRREL, N, ITMAX, XGUESS, X, FNORM,
   FVEC, FJAC, R, QTF, WK)
   ```

   The additional arguments are as follows:

   FVEC — A vector of length N. FVEC contains the functions evaluated at the point X.

FJAC — An N by N matrix. FJAC contains the orthogonal matrix Q produced by the QR factorization of the final approximate Jacobian.

R — A vector of length N * (N + 1)/2. R contains the upper triangular matrix produced by the QR factorization of the final approximate Jacobian. R is stored row-wise.

QTF — A vector of length N. QTF contains the vector TRANS(Q) * FVEC.

WK — A work vector of length 5 * N.

2. Informational errors

Type	Code	
4	1	The number of calls to FCN has exceeded ITMAX * (N + 1). A new initial guess may be tried.
4	2	ERRREL is too small. No further improvement in the approximate solution is possible.
4	3	The iteration has not made good progress. A new initial guess may be tried.

Description

Routine NEQNF is based on the MINPACK subroutine HYBRD1, which uses a modification of M.J.D. Powell's hybrid algorithm. This algorithm is a variation of Newton's method, which uses a finite-difference approximation to the Jacobian and takes precautions to avoid large step sizes or increasing residuals. For further description, see More et al. (1980).

Since a finite-difference method is used to estimate the Jacobian, for single precision calculation, the Jacobian may be so incorrect that the algorithm terminates far from a root. In such cases, high precision arithmetic is recommended. Also, whenever the exact Jacobian can be easily provided, IMSL routine NEQNJ (page 1165) should be used instead.

NEQNJ

Solves a system of nonlinear equations using a modified Powell hybrid algorithm with a user-supplied Jacobian.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the system of equations to be solved. The usage is CALL FCN (X, F, N), where

X – The point at which the functions are evaluated. (Input)
X should not be changed by FCN.

F – The computed function values at the point X. (Output)

N – Length of X, F. (Input)

FCN must be declared EXTERNAL in the calling program.

LSJAC — User-supplied SUBROUTINE to evaluate the Jacobian at a point X. The usage is CALL LSJAC (N, X, FJAC), where

N – Length of X. (Input)

X – The point at which the function is evaluated. (Input)
X should not be changed by LSJAC.

FJAC — The computed N by N Jacobian at the point X. (Output)

LSJAC must be declared EXTERNAL in the calling program.

X — A vector of length N. (Output)
X contains the best estimate of the root found by NEQNJ.

Optional Arguments

ERRREL — Stopping criterion. (Input)
The root is accepted if the relative error between two successive approximations to this root is less than ERRREL.
Default: ERRREL = 1.e-4 for single precision and 1.d-8 for double precision.

N — The number of equations to be solved and the number of unknowns. (Input)
Default: N = size (X,1).

ITMAX — The maximum allowable number of iterations. (Input)
Suggested value = 200.
Default: ITMAX = 200.

XGUESS — A vector of length N. (Input)
XGUESS contains the initial estimate of the root.
Default: XGUESS = 0.0.

FNORM — A scalar that has the value $F(1)^2 + \ldots + F(N)^2$ at the point X. (Output)

FORTRAN 90 Interface

Generic: CALL NEQNJ (FCN, LSJAC, X [,…])

Specific: The specific interface names are S_NEQNJ and D_NEQNJ.

FORTRAN 77 Interface

Single: CALL NEQNJ (FCN, LSJAC, ERRREL, N, ITMAX, XGUESS, X, FNORM)

Double: The double precision name is DNEQNJ.

Example

The following 3 × 3 system of nonlinear equations

$$
\begin{aligned}
f_1(x) &= x_1 + e^{x_1-1} + (x_2 + x_3)^2 - 27 = 0\\
f_2(x) &= e^{x_2-2}/x_1 + x_3^2 - 10 = 0\\
f_3(x) &= x_3 + \sin(x_2 - 2) + x_2^2 - 7 = 0
\end{aligned}
$$

is solved with the initial guess (4.0, 4.0, 4.0).

```
      USE NEQNJ_INT
      USE UMACH_INT
!                                 Declare variables
      INTEGER    N
      PARAMETER  (N=3)
!
      INTEGER    K, NOUT
      REAL       FNORM, X(N), XGUESS(N)
      EXTERNAL   FCN, LSJAC
!                                 Set values of initial guess
!                                 XGUESS = (  4.0  4.0  4.0  )
!
      DATA XGUESS/4.0, 4.0, 4.0/
!
!
      CALL UMACH (2, NOUT)
!                                 Find the solution
      CALL NEQNJ (FCN, LSJAC, X, XGUESS=XGUESS, FNORM=FNORM)
!                                 Output
      WRITE (NOUT,99999) (X(K),K=1,N), FNORM
99999 FORMAT (’  The roots found are’, /, ’  X = (’, 3F5.1, &
            ’)’, /, ’  with FNORM = ’,F5.4, //)
!
      END
!                                 User-supplied subroutine
      SUBROUTINE FCN (X, F, N)
      INTEGER    N
      REAL       X(N), F(N)
!
      REAL       EXP, SIN
      INTRINSIC  EXP, SIN
!
      F(1) = X(1) + EXP(X(1)-1.0) + (X(2)+X(3))*(X(2)+X(3)) - 27.0
      F(2) = EXP(X(2)-2.0)/X(1) + X(3)*X(3) - 10.0
      F(3) = X(3) + SIN(X(2)-2.0) + X(2)*X(2) - 7.0
      RETURN
      END
!                                 User-supplied subroutine to
!                                 compute Jacobian
      SUBROUTINE LSJAC (N, X, FJAC)
      INTEGER    N
      REAL       X(N), FJAC(N,N)
!
      REAL       COS, EXP
      INTRINSIC  COS, EXP
!
      FJAC(1,1) = 1.0 + EXP(X(1)-1.0)
      FJAC(1,2) = 2.0*(X(2)+X(3))
      FJAC(1,3) = 2.0*(X(2)+X(3))
      FJAC(2,1) = -EXP(X(2)-2.0)*(1.0/X(1)**2)
      FJAC(2,2) = EXP(X(2)-2.0)*(1.0/X(1))
      FJAC(2,3) = 2.0*X(3)
      FJAC(3,1) = 0.0
      FJAC(3,2) = COS(X(2)-2.0) + 2.0*X(2)
      FJAC(3,3) = 1.0
      RETURN
      END
```

Output

```
The roots found are
X = (  1.0  2.0  3.0)
with FNORM =.0000
```

Comments

1. Workspace may be explicitly provided, if desired, by use of N2QNJ/DN2QNJ. The reference is:

   CALL N2QNJ (FCN, LSJAC, ERRREL, N, ITMAX, XGUESS, X, FNORM, FVEC, FJAC, R, QTF, WK)

   The additional arguments are as follows:

   FVEC — A vector of length N. FVEC contains the functions evaluated at the point X.

   FJAC — An N by N matrix. FJAC contains the orthogonal matrix Q produced by the QR factorization of the final approximate Jacobian.

   R — A vector of length N * (N + 1)/2. R contains the upper triangular matrix produced by the QR factorization of the final approximate Jacobian. R is stored row-wise.

   QTF — A vector of length N. QTF contains the vector TRANS(Q) * FVEC.

   WK — A work vector of length 5 * N.

2. Informational errors

Type	Code	
4	1	The number of calls to FCN has exceeded ITMAX. A new initial guess may be tried.

4	2	ERRREL is too small. No further improvement in the approximate solution is possible.
4	3	The iteration has not made good progress. A new initial guess may be tried.

Description

Routine NEQNJ is based on the MINPACK subroutine HYBRDJ, which uses a modification of M.J.D. Powell's hybrid algorithm. This algorithm is a variation of Newton's method, which takes precautions to avoid large step sizes or increasing residuals. For further description, see More et al. (1980).

NEQBF

Solves a system of nonlinear equations using factored secant update with a finite-difference approximation to the Jacobian.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the system of equations to be solved. The usage is CALL FCN (N, X, F), where

N – Length of X and F. (Input)

X – The point at which the functions are evaluated. (Input)
X should not be changed by FCN.

F – The computed function values at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

X — Vector of length N containing the approximate solution. (Output)

Optional Arguments

N — Dimension of the problem. (Input)
Default: N = size (X,1).

XGUESS — Vector of length N containing initial guess of the root. (Input)
Default: XGUESS = 0.0.

XSCALE — Vector of length N containing the diagonal scaling matrix for the variables. (Input)
XSCALE is used mainly in scaling the distance between two points. In the absence of other information, set all entries to 1.0. If internal scaling is desired for XSCALE, set IPARAM (6) to 1.
Default: XSCALE = 1.0.

FSCALE — Vector of length N containing the diagonal scaling matrix for the functions. (Input)

FSCALE is used mainly in scaling the function residuals. In the absence of other information, set all entries to 1.0.
Default: FSCALE = 1.0.

IPARAM — Parameter vector of length 6. (Input/Output)
Set IPARAM (1) to zero for default values of IPARAM and RPARAM. See Comment 4.
Default: IPARAM = 0.

RPARAM — Parameter vector of length 5. (Input/Output)
See Comment 4.

FVEC — Vector of length N containing the values of the functions at the approximate solution. (Output)

FORTRAN 90 Interface

Generic: `CALL NEQBF (FCN, X [,…])`

Specific: The specific interface names are S_NEQBF and D_NEQBF.

FORTRAN 77 Interface

Single:
```
CALL NEQBF (FCN, N, XGUESS, XSCALE, FSCALE, IPARAM, RPARAM,
            X, FVEC)
```

Double: The double precision name is DNEQBF.

Example

The following 3 × 3 system of nonlinear equations:

$$f_1(x) = x_1 + e^{x_1-1} + (x_2 + x_3)^2 - 27 = 0$$
$$f_2(x) = e^{x_2-2} / x_1 + x_3^2 - 10 = 0$$
$$f_3(x) = x_3 + \sin(x_2 - 2) + x_2^2 - 7 = 0$$

is solved with the initial guess (4.0, 4.0, 4.0).

```
      USE NEQBF_INT
      USE UMACH_INT
!                                 Declare variables
      INTEGER    N
      PARAMETER  (N=3)
!
      INTEGER    K, NOUT
      REAL       X(N), XGUESS(N)
      EXTERNAL   FCN
!                                 Set values of initial guess
!                                 XGUESS = (  4.0  4.0  4.0 )
!
      DATA XGUESS/3*4.0/
!
!                                 Find the solution
      CALL NEQBF (FCN, X, XGUESS=XGUESS)
!                                 Output
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) (X(K),K=1,N)
99999 FORMAT ('  The solution to the system is', /, '  X = (', 3F8.3, &
            ')')
!
      END
!                                 User-defined subroutine
      SUBROUTINE FCN (N, X, F)
      INTEGER    N
      REAL       X(N), F(N)
!
      REAL       EXP, SIN
      INTRINSIC  EXP, SIN
!
      F(1) = X(1) + EXP(X(1)-1.0) + (X(2)+X(3))*(X(2)+X(3)) - 27.0
      F(2) = EXP(X(2)-2.0)/X(1) + X(3)*X(3) - 10.0
      F(3) = X(3) + SIN(X(2)-2.0) + X(2)*X(2) - 7.0
      RETURN
      END
```

Output

```
The solution to the system is
X = (   1.000   2.000   3.000)
```

Comments

1. Workspace may be explicitly provided, if desired, by use of N2QBF/DN2QBF. The reference is:

   ```
   CALL N2QBF (FCN, N, XGUESS, XSCALE, FSCALE, IPARAM,
   RPARAM, X, FVEC, WK, LWK)
   ```

   The additional arguments are as follows:

   WK — A work vector of length LWK. On output WK contains the following information:

   The third N locations contain the last step taken.

   The fourth N locations contain the last Newton step.

   The final N^2 locations contain an estimate of the Jacobian at the solution.

   LWK — Length of WK, which must be at least 2 * N^2 + 11 * N.   (Input)

2. Informational errors

Type	Code	
3	1	The last global step failed to decrease the 2-norm of F(X) sufficiently; either the current point is close to a root of F(X) and no more accuracy is possible, or the secant approximation to the Jacobian is inaccurate, or the step tolerance is too large.
3	3	The scaled distance between the last two steps is less than the step tolerance; the current point is probably an approximate root of F(X) (unless STEPTL is too large).
3	4	Maximum number of iterations exceeded.
3	5	Maximum number of function evaluations exceeded.
3	7	Five consecutive steps of length STEPMX have been taken; either the 2-norm of F(X) asymptotes from above to a finite value in some direction or the maximum allowable step size STEPMX is too small.

3. The stopping criterion for NEQBF occurs when the scaled norm of the functions is less than the scaled function tolerance (RPARAM(1)).

4. If the default parameters are desired for NEQBF, then set IPARAM(1) to zero and call routine NEQBF. Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM, then the following steps should be taken before calling NEQBF:

   CALL N4QBJ (IPARAM, RPARAM)

   Set nondefault values for desired IPARAM, RPARAM elements.

Note that the call to N4QBJ will set IPARAM and RPARAM to their default values, so only nondefault values need to be set above.

The following is a list of the parameters and the default values:

IPARAM — Integer vector of length 6.

IPARAM(1) = Initialization flag.

IPARAM(2) = Number of good digits in the function.
Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.
Default: 100.

IPARAM(4) = Maximum number of function evaluations.
Default: 400.

IPARAM(5) = Maximum number of Jacobian evaluations.
Default: not used in NEQBF.

IPARAM(6) = Internal variable scaling flag.
If IPARAM(6) = 1, then the values of XSCALE are set internally.
Default: 0.

RPARAM — Real vector of length 5.

RPARAM(1) = Scaled function tolerance.
The scaled norm of the functions is computed as

$$\max_i \left(\left|f_i\right| * fs_i\right)$$

where f_i is the i-th component of the function vector F, and fs_i is the i-th component of FSCALE.
Default:

$$\sqrt{\varepsilon}$$

where ε is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)
The scaled norm of the step between two points x and y is computed as

$$\max_i \left\{\frac{\left|x_i - y_i\right|}{\max\left(\left|x_i\right|, 1/s_i\right)}\right\}$$

where s_i is the i-th component of XSCALE.
Default: $\varepsilon^{2/3}$, where ε is the machine precision.

RPARAM(3) = False convergence tolerance.
Default: not used in NEQBF.

RPARAM(4) = Maximum allowable step size. (STEPMX)

Default: 1000 * max(ε_1, ε_2), where

$$\varepsilon_1 = \sqrt{\sum_{i=1}^{n} \left(s_i t_i\right)^2}$$

$\varepsilon_2 = \|s\|_2$, s = XSCALE, and t = XGUESS.

RPARAM(5) = Size of initial trust region.
Default: based on the initial scaled Cauchy step.

If double precision is desired, then DN4QBJ is called and RPARAM is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to “Error Handling” in the Introduction.

Description

Routine NEQBF uses a secant algorithm to solve a system of nonlinear equations, i.e.,

$$F(x) = 0$$

where $F : \mathbf{R}^n \to \mathbf{R}^n$, and $x \in \mathbf{R}^n$.

From a current point, the algorithm uses a double dogleg method to solve the following subproblem approximately:

$$\min_{s \in \mathbf{R}^n} \left\| F(x_c) + J(x_c)s \right\|_2$$

$$\text{subject to } \| s \|_2 \le \delta_c$$

to get a direction s_c, where $F(x_c)$ and $J(x_c)$ are the function values and the approximate Jacobian respectively evaluated at the current point x_c. Then, the function values at the point $x_n = x_c + s_c$ are evaluated and used to decide whether the new point x_n should be accepted.

When the point x_n is rejected, this routine reduces the trust region δ_c and goes back to solve the subproblem again. This procedure is repeated until a better point is found.

The algorithm terminates if the new point satisfies the stopping criterion. Otherwise, δ_c is adjusted, and the approximate Jacobian is updated by Broyden’s formula,

$$J_n = J_c + \frac{(y - J_c s_c) s_c^T}{s_c^T s_c}$$

where $J_n = J(x_n)$, $J_c = J(x_c)$, and $y = F(x_n) - F(x_c)$. The algorithm then continues using the new point as the current point, i.e. $x_c \leftarrow x_n$.

For more details, see Dennis and Schnabel (1983, Chapter 8).

Since a finite-difference method is used to estimate the initial Jacobian, for single precision calculation, the Jacobian may be so incorrect that the algorithm terminates far from a root. In such cases, high precision arithmetic is recommended. Also, whenever the exact Jacobian can be easily provided, IMSL routine NEQBJ (page 1174) should be used instead.

NEQBJ

Solves a system of nonlinear equations using factored secant update with a user-supplied Jacobian.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the system of equations to be solved. The usage is CALL FCN (N, X, F), where

N – Length of X and F. (Input)

X – The point at which the functions are evaluated. (Input)
X should not be changed by FCN.
F – The computed function values at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

JAC — User-supplied SUBROUTINE to evaluate the Jacobian at a point X. The usage is CALL JAC (N, X, FJAC, LDFJAC), where

N – Length of X. (Input)
X – Vector of length N at which point the Jacobian is evaluated. (Input)
X should not be changed by JAC.
FJAC – The computed N by N Jacobian at the point X. (Output)
LDFJAC – Leading dimension of FJAC. (Input)

JAC must be declared EXTERNAL in the calling program.

X — Vector of length N containing the approximate solution. (Output)

Optional Arguments

N — Dimension of the problem. (Input)
Default: N = size (X,1).

XGUESS — Vector of length N containing initial guess of the root. (Input)
Default: XGUESS = 0.0.

XSCALE — Vector of length N containing the diagonal scaling matrix for the variables. (Input)
XSCALE is used mainly in scaling the distance between two points. In the absence of other information, set all entries to 1.0. If internal scaling is desired for XSCALE, set IPARAM(6) to 1.
Default: XSCALE = 1.0.

FSCALE — Vector of length N containing the diagonal scaling matrix for the functions. (Input)
FSCALE is used mainly in scaling the function residuals. In the absence of other information, set all entries to 1.0.
Default: FSCALE = 1.0.

IPARAM — Parameter vector of length 6. (Input/Output)
Set IPARAM (1) to zero for default values of IPARAM and RPARAM.
See Comment 4.
Default: IPARAM = 0.

RPARAM — Parameter vector of length 5. (Input/Output)
See Comment 4.

FVEC — Vector of length N containing the values of the functions at the approximate solution. (Output)

FORTRAN 90 Interface

Generic: `CALL NEQBJ (FCN, JAC, X [,…])`

Specific: The specific interface names are `S_NEQBJ` and `D_NEQBJ`.

FORTRAN 77 Interface

Single:

```
CALL NEQBJ (FCN, JAC, N, XGUESS, XSCALE, FSCALE, IPARAM,
RPARAM, X, FVEC)
```

Double: The double precision name is `DNEQBJ`.

Example

The following 3 × 3 system of nonlinear equations

$$f_1(x) = x_1 + e^{x_1 - 1} + (x_2 + x_3)^2 - 27 = 0$$
$$f_2(x) = e^{x_2 - 2} / x_1 + x_3^2 - 10 = 0$$
$$f_3(x) = x_3 + \sin(x_2 - 2) + x_2^2 - 7 = 0$$

is solved with the initial guess (4.0, 4.0, 4.0).

```
      USE NEQBJ_INT
      USE UMACH_INT
!                                 Declare variables
      INTEGER    N
      PARAMETER  (N=3)
!
      INTEGER    K, NOUT
      REAL       X(N), XGUESS(N)
      EXTERNAL   FCN, JAC
!                                 Set values of initial guess
!                                 XGUESS = (  4.0  4.0  4.0 )
!
      DATA XGUESS/3*4.0/
!                                 Find the solution
      CALL NEQBJ (FCN, JAC, X, XGUESS=XGUESS)
!                                 Output
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) (X(K),K=1,N)
99999 FORMAT ('  The solution to the system is', /, '  X = (', 3F8.3, &
            ')')
!
      END
!                                 User-defined subroutine
      SUBROUTINE FCN (N, X, F)
      INTEGER    N
      REAL       X(N), F(N)
!
      REAL       EXP, SIN
      INTRINSIC  EXP, SIN
```

```
!
      F(1) = X(1) + EXP(X(1)-1.0) + (X(2)+X(3))*(X(2)+X(3)) - 27.0
      F(2) = EXP(X(2)-2.0)/X(1) + X(3)*X(3) - 10.0
      F(3) = X(3) + SIN(X(2)-2.0) + X(2)*X(2) - 7.0
      RETURN
      END
!                                 User-supplied subroutine to
!                                 compute Jacobian
      SUBROUTINE JAC (N, X, FJAC, LDFJAC)
      INTEGER    N, LDFJAC
      REAL       X(N), FJAC(LDFJAC,N)
!
      REAL       COS, EXP
      INTRINSIC  COS, EXP
!
      FJAC(1,1) = 1.0 + EXP(X(1)-1.0)
      FJAC(1,2) = 2.0*(X(2)+X(3))
      FJAC(1,3) = 2.0*(X(2)+X(3))
      FJAC(2,1) = -EXP(X(2)-2.0)*(1.0/X(1)**2)
      FJAC(2,2) = EXP(X(2)-2.0)*(1.0/X(1))
      FJAC(2,3) = 2.0*X(3)
      FJAC(3,1) = 0.0
      FJAC(3,2) = COS(X(2)-2.0) + 2.0*X(2)
      FJAC(3,3) = 1.0
      RETURN
      END
```

Output

```
The solution to the system is
X = (   1.000   2.000   3.000)
```

Comments

1. Workspace may be explicitly provided, if desired, by use of N2QBJ/DN2QBJ. The reference is:

   ```
   CALL N2QBJ (FCN, JAC, N, XGUESS, XSCALE, FSCALE,
   IPARAM, RPARAM, X, FVEC, WK, LWK)
   ```

   The additional arguments are as follows:

   WK — A work vector of length LWK. On output WK contains the following information: The third N locations contain the last step taken. The fourth N locations contain the last Newton step. The final N^2 locations contain an estimate of the Jacobian at the solution.

   LWK — Length of WK, which must be at least $2 * N^2 + 11 * N$. (Input)

2. Informational errors

Type	Code	
3	1	The last global step failed to decrease the 2-norm of F(X) sufficiently; either the current point is close to a root of F(X) and no more

		accuracy is possible, or the secant approximation to the Jacobian is inaccurate, or the step tolerance is too large.
3	3	The scaled distance between the last two steps is less than the step tolerance; the current point is probably an approximate root of F(X) (unless STEPTL is too large).
3	4	Maximum number of iterations exceeded.
3	5	Maximum number of function evaluations exceeded.
3	7	Five consecutive steps of length STEPMX have been taken; either the 2-norm of F(X) asymptotes from above to a finite value in some direction or the maximum allowable stepsize STEPMX is too small.

3. The stopping criterion for NEQBJ occurs when the scaled norm of the functions is less than the scaled function tolerance (RPARAM(1)).

4. If the default parameters are desired for NEQBJ, then set IPARAM(1) to zero and call routine NEQBJ. Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM, then the following steps should be taken before calling NEQBJ:

   CALL N4QBJ (IPARAM, RPARAM)

   Set nondefault values for desired IPARAM, RPARAM elements.

   Note that the call to N4QBJ will set IPARAM and RPARAM to their default values, so only nondefault values need to be set above.

   The following is a list of the parameters and the default values:

   IPARAM — Integer vector of length 6.

   IPARAM(1) = Initialization flag.

   IPARAM(2) = Number of good digits in the function.
   Default: Machine dependent.

   IPARAM(3) = Maximum number of iterations.
   Default: 100.

   IPARAM(4) = Maximum number of function evaluations.
   Default: 400.

   IPARAM(5) = Maximum number of Jacobian evaluations.
   Default: not used in NEQBJ.

   IPARAM(6) = Internal variable scaling flag.
   If IPARAM(6) = 1, then the values of XSCALE are set internally.
   Default: 0.

   RPARAM — Real vector of length 5.

RPARAM(1) = Scaled function tolerance.
The scaled norm of the functions is computed as

$$\max_i \left(\left| f_i \right| * fs_i \right)$$

where f_i is the i-th component of the function vector F, and fs_i is the i-th component of FSCALE.
Default:

$$\sqrt{\varepsilon}$$

where ε is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)
The scaled norm of the step between two points x and y is computed as

$$\max_i \left\{ \frac{\left| x_i - y_i \right|}{\max\left(\left| x_i \right|, 1/s_i \right)} \right\}$$

where s_i is the i-th component of XSCALE.

Default: $\varepsilon^{2/3}$, where ε is the machine precision.

RPARAM(3) = False convergence tolerance.
Default: not used in NEQBJ.

RPARAM(4) = Maximum allowable step size. (STEPMX)

Default: 1000 * max(ε_1, ε_2), where

$$\varepsilon_1 = \sqrt{\sum_{i=1}^{n} \left(s_i t_i \right)^2}$$

$\varepsilon_2 = \|s\|_2$, s = XSCALE, and t = XGUESS.

RPARAM(5) = Size of initial trust region.
Default: based on the initial scaled Cauchy step.

If double precision is desired, then DN4QBJ is called and RPARAM is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to “Error Handling” in the Introduction.

Description

Routine NEQBJ uses a secant algorithm to solve a system of nonlinear equations, i. e.,

$$F(x) = 0$$

where $F : \mathbf{R}^n \rightarrow \mathbf{R}^n$, and $x \in \mathbf{R}^n$.

From a current point, the algorithm uses a double dogleg method to solve the following subproblem approximately:

$$\min_{s \in \mathbf{R}^n} \left\| F\left(x_c\right) + J\left(x_c\right) s \right\|_2$$

$$\text{subject to } \|s\|_2 \leq \delta_c$$

to get a direction s_c, where $F(x_c)$ and $J(x_c)$ are the function values and the approximate Jacobian respectively evaluated at the current point x_c. Then, the function values at the point $x_n = x_c + s_c$ are evaluated and used to decide whether the new point x_n should be accepted.

When the point x_n is rejected, this routine reduces the trust region δ_c and goes back to solve the subproblem again. This procedure is repeated until a better point is found.

The algorithm terminates if the new point satisfies the stopping criterion. Otherwise, δ_c is adjusted, and the approximate Jacobian is updated by Broyden's formula,

$$J_n = J_c + \frac{\left(y - J_c s_c\right) s_c^T}{s_c^T s_c}$$

where $J_n = J(x_n)$, $J_c = J(x_c)$, and $y = F(x_n) - F(x_c)$. The algorithm then continues using the new point as the current point, i.e. $x_c \leftarrow x_n$.

For more details, see Dennis and Schnabel (1983, Chapter 8).

Chapter 8: Optimization

Routines

Usage Notes

Unconstrained Minimization

The unconstrained minimization problem can be stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

where f : $\mathbf{R}^n \rightarrow \mathbf{R}$ is at least continuous. The routines for unconstrained minimization are grouped into three categories: univariate functions (`UV***`), multivariate functions (`UM***`), and nonlinear least squares (`UNLS*`).

For the univariate function routines, it is assumed that the function is unimodal within the specified interval. Otherwise, only a local minimum can be expected. For further discussion on unimodality, see Brent (1973).

A quasi-Newton method is used for the multivariate function routines `UMINF` (page 1196) and `UMING` (page 1202), whereas `UMIDH` (page 1208) and `UMIAH` (page 1213) use a modified Newton algorithm. The routines `UMCGF` (page 1219) and `UMCGG` (page 1223) make use of a conjugate gradient approach, and `UMPOL` (page 1227) uses a polytope method. For more details on these algorithms, see the documentation for the corresponding routines.

The nonlinear least squares routines use a modified Levenberg-Marquardt algorithm. If the nonlinear least squares problem is a nonlinear data-fitting problem, then software that is designed to deliver better statistical output may be useful; see IMSL (1991).

These routines are designed to find only a local minimum point. However, a function may have many local minima. It is often possible to obtain a better local solution by trying different initial points and intervals.

High precision arithmetic is recommended for the routines that use only function values. Also it is advised that the derivative-checking routines `CH***` be used to ensure the accuracy of the user-supplied derivative evaluation subroutines.

Minimization with Simple Bounds

The minimization with simple bounds problem can be stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

$$\text{subject to } l_i \le x_i \le u_i, \text{ for } i = 1, 2, \ldots, n$$

where $f : \mathbf{R}^n \rightarrow \mathbf{R}$, and all the variables are not necessarily bounded.

The routines BCO** use the same algorithms as the routines UMI**, and the routines BCLS* are the corresponding routines of UNLS*. The only difference is that an active set strategy is used to ensure that each variable stays within its bounds. The routine BCPOL (page 1271) uses a function comparison method similar to the one used by UMPOL (page 1227). Convergence for these polytope methods is not guaranteed; therefore, these routines should be used as a last alternative.

Linearly Constrained Minimization

The linearly constrained minimization problem can be stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

$$\text{subject to } Ax = b$$

where $f : \mathbf{R}^n \rightarrow \mathbf{R}$, A is an $m \times n$ coefficient matrix, and b is a vector of length m. If $f(x)$ is linear, then the problem is a linear programming problem; if $f(x)$ is quadratic, the problem is a quadratic programming problem.

The routine DLPRS (page 1297) uses a revised simplex method to solve small- to medium-sized linear programming problems. No sparsity is assumed since the coefficients are stored in full matrix form.

The routine QPROG (page 1307) is designed to solve convex quadratic programming problems using a dual quadratic programming algorithm. If the given Hessian is not positive definite, then QPROG modifies it to be positive definite. In this case, output should be interpreted with care.

The routines LCONF (page 1310) and LCONG (page 1316) use an iterative method to solve the linearly constrained problem with a general objective function. For a detailed description of the algorithm, see Powell (1988, 1989).

Nonlinearly Constrained Minimization

The nonlinearly constrained minimization problem can be stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

$$\begin{aligned} \text{subject to } g_i(x) &= 0, \text{ for } && i = 1, 2, \ldots, m_1 \\ g_i(x) &\ge 0, \text{ for } && i = m_1 + 1, \ldots, m \end{aligned}$$

where $f : \mathbf{R}^n \rightarrow \mathbf{R}$ and $g_i : \mathbf{R}^n \rightarrow \mathbf{R}$, for $i = 1, 2, \ldots, m$

The routines NNLPF (page 1323) and NNLPG (page 1329) use a sequential equality constrained quadratic programming method. A more complete discussion of this algorithm can be found in the documentation.

Selection of Routines

The following general guidelines are provided to aid in the selection of the appropriate routine.

Unconstrained Minimization

1. For the univariate case, use UVMID (page 1189) when the gradient is available, and use UVMIF (page 1182) when it is not. If discontinuities exist, then use UVMGS (page 1193).

2. For the multivariate case, use UMCG* when storage is a problem, and use UMPOL (page 1227) when the function is nonsmooth. Otherwise, use UMI** depending on the availability of the gradient and the Hessian.

3. For least squares problems, use UNLSJ (page 1237) when the Jacobian is available, and use UNLSF (page 1231) when it is not.

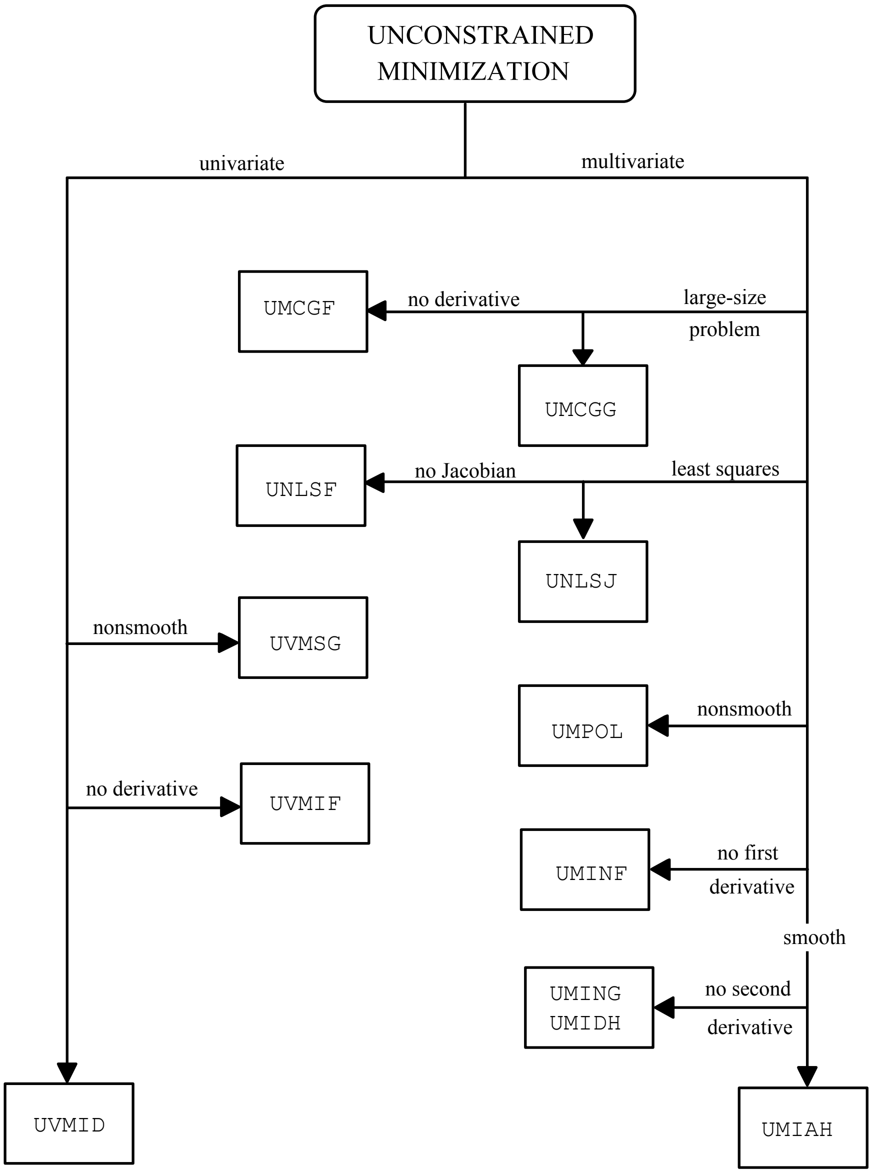
Minimization with Simple Bounds

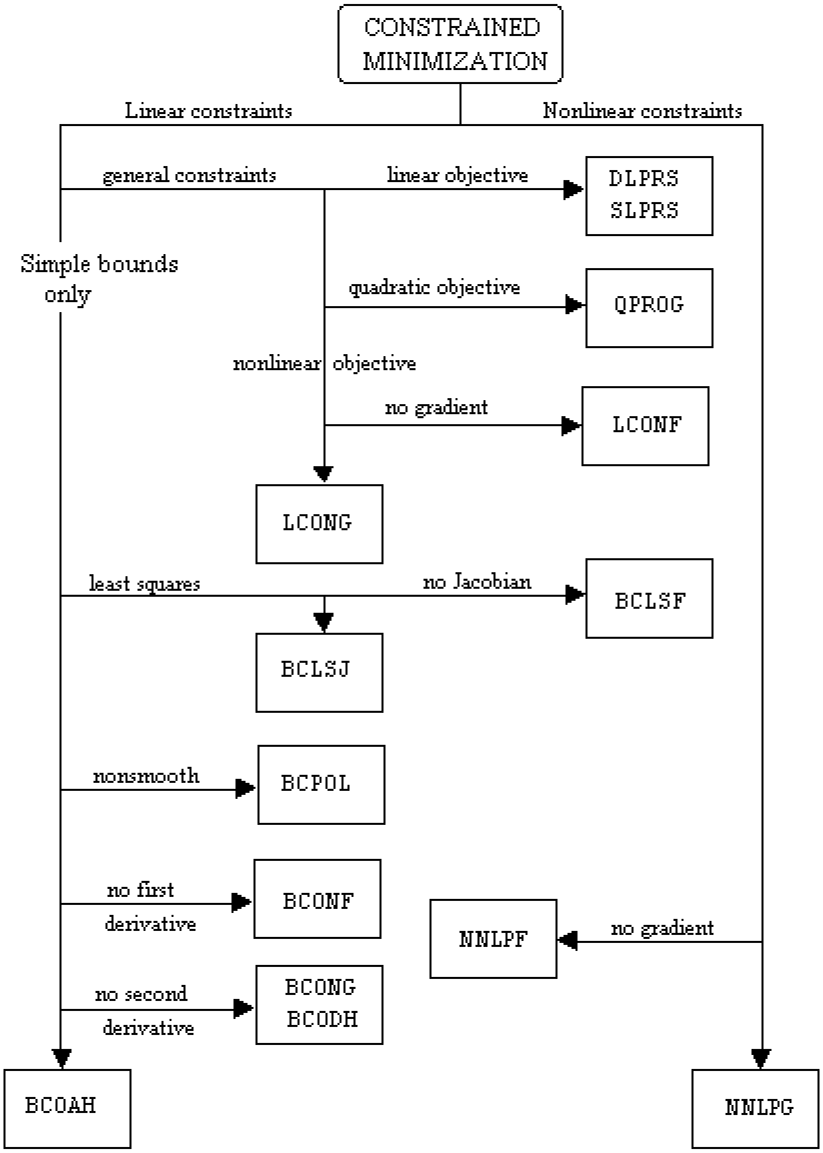
1. Use BCONF (page 1243) when only function values are available. When first derivatives are available, use either BCONG (page 1249) or BCODH (page 1257). If first and second derivatives are available, then use BCOAH (page 1263).

2. For least squares, use BCLSF (page 1274) or BCLSJ (page 1281) depending on the availability of the Jacobian.

3. Use BCPOL (page 1271) for nonsmooth functions that could not be solved satisfactorily by the other routines.

The following charts provide a quick reference to routines in this chapter:





UVMIF

Finds the minimum point of a smooth function of a single variable using only function evaluations.

Required Arguments

F — User-supplied `FUNCTION` to compute the value of the function to be minimized. The form is `F`(`X`), where
`X` – The point at which the function is evaluated. (Input)

X should not be changed by F.
F – The computed function value at the point X. (Output)

F must be declared EXTERNAL in the calling program.

XGUESS — An initial guess of the minimum point of F. (Input)

BOUND — A positive number that limits the amount by which X may be changed from its initial value. (Input)

X — The point at which a minimum value of F is found. (Output)

Optional Arguments

STEP — An order of magnitude estimate of the required change in X. (Input)
Default: STEP = 1.0.

XACC — The required absolute accuracy in the final value of X. (Input)
On a normal return there are points on either side of X within a distance XACC at which F is no less than F(X).
Default: XACC = 1.e-4.

MAXFN — Maximum number of function evaluations allowed. (Input)
Default: MAXFN = 1000.

FORTRAN 90 Interface

Generic: CALL UVMIF (F, XGUESS, BOUND, X [,…])

Specific: The specific interface names are S_UVMIF and D_UVMIF.

FORTRAN 77 Interface

Single: CALL UVMIF (F, XGUESS, STEP, BOUND, XACC, MAXFN, X)

Double: The double precision name is DUVMIF.

Example

A minimum point of $e^x - 5x$ is found.

```
      USE UVMIF_INT
      USE UMACH_INT
!                                  Declare variables
      INTEGER    MAXFN, NOUT
      REAL       BOUND, F, FX, STEP, X, XACC, XGUESS
      EXTERNAL   F
!                                  Initialize variables
```

```
      XGUESS = 0.0
      XACC   = 0.001
      BOUND  = 100.0
      STEP   = 0.1
      MAXFN  = 50
!
!                                  Find minimum for F = EXP(X) - 5X
      CALL UVMIF (F, XGUESS, BOUND, X, STEP=STEP, XACC=XACC, MAXFN=MAXFN)
      FX = F(X)
!                                  Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) X, FX
!
99999 FORMAT (’   The minimum is at ’, 7X, F7.3, //, ’   The function ’ &
            , ’value is ’, F7.3)
!
      END
!                                  Real function: F = EXP(X) - 5.0*X
      REAL FUNCTION F (X)
      REAL       X
!
      REAL       EXP
      INTRINSIC  EXP
!
      F = EXP(X) - 5.0E0*X
!
      RETURN
      END
```

Output

```
The minimum is at          1.609

The function value is  -3.047
```

Comments

Informational errors

Type	Code	
3	1	Computer rounding errors prevent further refinement of X.
3	2	The final value of X is at a bound. The minimum is probably beyond the bound.
4	3	The number of function evaluations has exceeded MAXFN.

Description

The routine UVMIF uses a safeguarded quadratic interpolation method to find a minimum point of a univariate function. Both the code and the underlying algorithm are based on the routine ZXLSF written by M.J.D. Powell at the University of Cambridge.

The routine UVMIF finds the least value of a univariate function, *f*, that is specified by the function subroutine F. Other required data include an initial estimate of the solution, XGUESS , and a positive number BOUND. Let x_0 = XGUESS and b = BOUND, then x is restricted to the interval $[x_0 - b, x_0 + b]$. Usually, the algorithm begins the search by moving from x_0 to $x = x_0 + s$, where s = `STEP` is also provided by the user and may be positive or negative. The first two function evaluations indicate the direction to the minimum point, and the search strides out along this direction until a bracket on a minimum point is found or until x reaches one of the bounds $x_0 \pm b$. During this stage, the step length increases by a factor of between two and nine per function evaluation; the factor depends on the position of the minimum point that is predicted by quadratic interpolation of the three most recent function values.

When an interval containing a solution has been found, we will have three points, x_1, x_2, and x_3, with $x_1 < x_2 < x_3$ and $f(x_2) \le f(x_1)$ and $f(x_2) \le f(x_3)$. There are three main ingredients in the technique for choosing the new x from these three points. They are (i) the estimate of the minimum point that is given by quadratic interpolation of the three function values, (ii) a tolerance parameter ε, that depends on the closeness of f to a quadratic, and (iii) whether x_2 is near the center of the range between x_1 and x_3 or is relatively close to an end of this range. In outline, the new value of x is as near as possible to the predicted minimum point, subject to being at least ε from x_2, and subject to being in the longer interval between x_1 and x_2 or x_2 and x_3 when x_2 is particularly close to x_1 or x_3. There is some elaboration, however, when the distance between these points is close to the required accuracy; when the distance is close to the machine precision; or when ε is relatively large.

The algorithm is intended to provide fast convergence when f has a positive and continuous second derivative at the minimum and to avoid gross inefficiencies in pathological cases, such as

$$f(x) = x + 1.001|x|$$

The algorithm can make ε large automatically in the pathological cases. In this case, it is usual for a new value of x to be at the midpoint of the longer interval that is adjacent to the least calculated function value. The midpoint strategy is used frequently when changes to f are dominated by computer rounding errors, which will almost certainly happen if the user requests an accuracy that is less than the square root of the machine precision. In such cases, the routine claims to have achieved the required accuracy if it knows that there is a local minimum point within distance δ of x, where δ = `XACC`, even though the rounding errors in f may cause the existence of other local minimum points nearby. This difficulty is inevitable in minimization routines that use only function values, so high precision arithmetic is recommended.

UVMID

Finds the minimum point of a smooth function of a single variable using both function evaluations and first derivative evaluations.

Required Arguments

F — User-supplied `FUNCTION` to define the function to be minimized. The form is `F(X)`, where

X — The point at which the function is to be evaluated. (Input)

F — The computed value of the function at X. (Output)

F must be declared EXTERNAL in the calling program.

G — User-supplied FUNCTION to compute the derivative of the function. The form is G(X), where

X — The point at which the derivative is to be computed. (Input)

G — The computed value of the derivative at X. (Output)

G must be declared EXTERNAL in the calling program.

A — A is the lower endpoint of the interval in which the minimum point of F is to be located. (Input)

B — B is the upper endpoint of the interval in which the minimum point of F is to be located. (Input)

X — The point at which a minimum value of F is found. (Output)

Optional Arguments

XGUESS — An initial guess of the minimum point of F. (Input)
Default: XGUESS = (a + b) / 2.0.

ERRREL — The required relative accuracy in the final value of X. (Input)
This is the first stopping criterion. On a normal return, the solution X is in an interval that contains a local minimum and is less than or equal to MAX(1.0, ABS(X)) * ERRREL. When the given ERRREL is less than machine epsilon, SQRT(machine epsilon) is used as ERRREL.
Default: ERRREL = 1.e-4.

GTOL — The derivative tolerance used to decide if the current point is a local minimum. (Input)
This is the second stopping criterion. X is returned as a solution when GX is less than or equal to GTOL. GTOL should be nonnegative, otherwise zero would be used.
Default: GTOL = 1.e-4.

MAXFN — Maximum number of function evaluations allowed. (Input)
Default: MAXFN = 1000.

FX — The function value at point X. (Output)

GX — The derivative value at point X. (Output)

FORTRAN 90 Interface

Generic: CALL UVMID (F, G, A, B, X [,…])

Specific: The specific interface names are S_UVMID and D_UVMID.

FORTRAN 77 Interface

Single: CALL UVMID (F, G, XGUESS, ERRREL, GTOL, MAXFN, A, B, X, FX, GX)

Double: The double precision name is DUVMID.

Example

A minimum point of $e^x - 5x$ is found.

```
      USE UVMID_INT
      USE UMACH_INT
!                                  Declare variables
      INTEGER    MAXFN, NOUT
      REAL       A, B, ERRREL, F, FX, G, GTOL, GX, X, XGUESS
      EXTERNAL   F, G
!                                  Initialize variables
      XGUESS = 0.0
!                                 Set ERRREL to zero in order
!                                 to use SQRT(machine epsilon)
!                                 as relative error
      ERRREL = 0.0
      GTOL   = 0.0
      A      = -10.0
      B      = 10.0
      MAXFN  = 50
!
!                                 Find minimum for F = EXP(X) - 5X
      CALL UVMID (F, G, A, B, X, XGUESS=XGUESS, ERRREL=ERRREL,   &
                 GTOL=FTOL, MAXFN=MAXFN, FX=FX, GX=GX)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) X, FX, GX
!
99999 FORMAT (’   The minimum is at ’, 7X, F7.3, //, ’   The function ’ &
            , ’value is ’, F7.3, //, ’   The derivative is ’, F7.3)
!
      END
!                                 Real function: F = EXP(X) - 5.0*X
      REAL FUNCTION F (X)
      REAL       X
!
      REAL       EXP
      INTRINSIC  EXP
!
      F = EXP(X) - 5.0E0*X
```

```
!
      RETURN
      END
!
      REAL FUNCTION G (X)
      REAL       X
!
      REAL       EXP
      INTRINSIC  EXP
!
      G = EXP(X) - 5.0E0
      RETURN
      END
```

Output

```
The minimum is at       1.609

The function value is  -3.047

The derivative is  -0.001
```

Comments

Informational errors

Type	Code	
3	1	The final value of X is at the lower bound. The minimum is probably beyond the bound.
3	2	The final value of X is at the upper bound. The minimum is probably beyond the bound.
4	3	The maximum number of function evaluations has been exceeded.

Description

The routine UVMID uses a descent method with either the secant method or cubic interpolation to find a minimum point of a univariate function. It starts with an initial guess and two endpoints. If any of the three points is a local minimum point and has least function value, the routine terminates with a solution. Otherwise, the point with least function value will be used as the starting point.

From the starting point, say x_c, the function value $f_c = f(x_c)$, the derivative value $g_c = g(x_c)$, and a new point x_n defined by $x_n = x_c - g_c$ are computed. The function $f_n = f(x_n)$, and the derivative $g_n = g(x_n)$ are then evaluated. If either $f_n \geq f_c$ or g_n has the opposite sign of g_c, then there exists a minimum point between x_c and x_n; and an initial interval is obtained. Otherwise, since x_c is kept as the point that has lowest function value, an interchange between x_n and x_c is performed. The secant method is then used to get a new point

$$x_s = x_c - g_c\left(\frac{g_n - g_c}{x_n - x_c}\right)$$

Let $x_n \leftarrow x_s$ and repeat this process until an interval containing a minimum is found or one of the convergence criteria is satisfied. The convergence criteria are as follows: Criterion 1:

$$\left|x_c - x_n\right| \le \varepsilon_c$$

Criterion 2:

$$\left|g_c\right| \le \varepsilon_g$$

where $\varepsilon_c = \max\{1.0, |x_c|\}\varepsilon$, ε is a relative error tolerance and ε_g is a gradient tolerance.

When convergence is not achieved, a cubic interpolation is performed to obtain a new point. Function and derivative are then evaluated at that point; and accordingly, a smaller interval that contains a minimum point is chosen. A safeguarded method is used to ensure that the interval reduces by at least a fraction of the previous interval. Another cubic interpolation is then performed, and this procedure is repeated until one of the stopping criteria is met.

UVMGS

Finds the minimum point of a nonsmooth function of a single variable.

Required Arguments

F — User-supplied `FUNCTION` to compute the value of the function to be minimized. The form is `F`(`X`), where

`X` – The point at which the function is evaluated. (Input)
`X` should not be changed by `F`.

F – The computed function value at the point `X`. (Output)

`F` must be declared `EXTERNAL` in the calling program.

A — On input, `A` is the lower endpoint of the interval in which the minimum of `F` is to be located. On output, `A` is the lower endpoint of the interval in which the minimum of `F` is located. (Input/Output)

B — On input, `B` is the upper endpoint of the interval in which the minimum of `F` is to be located. On output, `B` is the upper endpoint of the interval in which the minimum of `F` is located. (Input/Output)

XMIN — The approximate minimum point of the function `F` on the original interval (`A`, `B`). (Output)

Optional Arguments

TOL — The allowable length of the final subinterval containing the minimum point. (Input)
Default: `TOL` = 1.e-4.

FORTRAN 90 Interface

Generic: `CALL UVMGS (F, A, B, XMIN [,…])`

Specific: The specific interface names are `S_UVMGS` and `D_UVMGS`.

FORTRAN 77 Interface

Single: `CALL UVMGS (F, A, B, TOL, XMIN)`

Double: The double precision name is `DUVMGS`.

Example

A minimum point of $3x^2 - 2x + 4$ is found.

```
      USE UVMGS_INT
      USE UMACH_INT
!                                 Specification of variables
      INTEGER    NOUT
      REAL       A, B, FCN, FMIN, TOL, XMIN
      EXTERNAL   FCN
!                                 Initialize variables
      A   = 0.0E0
      B   = 5.0E0
      TOL = 1.0E-3
!                                 Minimize FCN
      CALL UVMGS (FCN, A, B, XMIN, TOL=TOL)
      FMIN = FCN(XMIN)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) XMIN, FMIN, A, B
99999 FORMAT ('     The minimum is at ', F5.3, //, '   The ', &
            'function value is ', F5.3, //, '   The final ', &
            'interval is (', F6.4, ',', F6.4, ')', /)
!
      END
!
!                                 REAL FUNCTION: F = 3*X**2 - 2*X + 4
      REAL FUNCTION FCN (X)
      REAL       X
!
      FCN = 3.0E0*X*X - 2.0E0*X + 4.0E0
!
      RETURN
      END
```

Output

```
The minimum is at 0.333

The function value is 3.667

The final interval is (0.3331,0.3340)
```

Comments

1. Informational errors

Type	Code	
3	1	TOL is too small to be satisfied.
4	2	Due to rounding errors F does not appear to be unimodal.

2. On exit from UVMGS without any error messages, the following conditions hold: (B-A) ≤ TOL.
A ≤ XMIN and XMIN ≤ B
F(XMIN) ≤ F(A) and F(XMIN) ≤ F(B)

3. On exit from UVMGS with error code 2, the following conditions hold:
A ≤ XMIN and XMIN ≤ B
F(XMIN) ≥ F(A) and F(XMIN) ≥ F(B) (only one equality can hold).
Further analysis of the function F is necessary in order to determine whether it is not unimodal in the mathematical sense or whether it appears to be not unimodal to the routine due to rounding errors in which case the A, B, and XMIN returned may be acceptable.

Description

The routine UVMGS uses the *golden section search* technique to compute to the desired accuracy the independent variable value that minimizes a unimodal function of one independent variable, where a known finite interval contains the minimum.

Let τ = TOL. The number of iterations required to compute the minimizing value to accuracy τ is the greatest integer less than or equal to

$$\frac{\ln\left(\tau/\left(b-a\right)\right)}{\ln\left(1-c\right)}+1$$

where a and b define the interval and

$$c=\left(3-\sqrt{5}\right)/2$$

The first two test points are v_1 and v_2 that are defined as

$$v_1 = a + c(b - a),\text{ and } v_2 = b - c(b - a)$$

If $f(v_1) < f(v_2)$, then the minimizing value is in the interval (a, v_2). In this case, $b \leftarrow v_2$, $v_2 \leftarrow v_1$, and $v_1 \leftarrow a + c(b - a)$. If $f(v_1) \geq f(v_2)$, the minimizing value is in (v_1, b). In this case, $a \leftarrow v_1$, $v_1 \leftarrow v_2$, and $v_2 \leftarrow b - c(b - a)$.

The algorithm continues in an analogous manner where only one new test point is computed at each step. This process continues until the desired accuracy τ is achieved. XMIN is set to the point producing the minimum value for the current iteration.

Mathematically, the algorithm always produces the minimizing value to the desired accuracy; however, numerical problems may be encountered. If f is too flat in part of the region of interest, the function may appear to be constant to the computer in that region. Error code 2 indicates that this problem has occurred. The user may rectify the problem by relaxing the requirement on τ, modifying (scaling, etc.) the form of f or executing the program in a higher precision.

UMINF

Minimizes a function of N variables using a quasi-Newton method and a finite-difference gradient.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function to be minimized. The usage is CALL FCN (N, X, F), where

N – Length of X. (Input)

X – The point at which the function is evaluated. (Input)
X should not be changed by FCN.

F – The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

X — Vector of length N containing the computed solution. (Output)

Optional Arguments

N — Dimension of the problem. (Input)
Default: N = size (X,1).

XGUESS — Vector of length N containing an initial guess of the computed solution. (Input)
Default: XGUESS = 0.0.

XSCALE — Vector of length N containing the diagonal scaling matrix for the variables. (Input)
XSCALE is used mainly in scaling the gradient and the distance between two points. In the absence of other information, set all entries to 1.0.
Default: XSCALE = 1.0.

FSCALE — Scalar containing the function scaling. (Input)
FSCALE is used mainly in scaling the gradient. In the absence of other information, set FSCALE to 1.0.
Default: FSCALE = 1.0.

IPARAM — Parameter vector of length 7. (Input/Output)
Set IPARAM(1) to zero for default values of IPARAM and RPARAM. See Comment 4.
Default: IPARAM = 0.

RPARAM — Parameter vector of length 7.(Input/Output)
See Comment 4.

FVALUE — Scalar containing the value of the function at the computed solution. (Output)

FORTRAN 90 Interface

Generic: CALL UMINF (FCN, X [,…])

Specific: The specific interface names are S_UMINF and D_UMINF.

FORTRAN 77 Interface

Single: CALL UMINF (FCN, N, XGUESS, XSCALE, FSCALE, IPARAM, RPARAM, X, FVALUE)

Double: The double precision name is DUMINF.

Example

The function

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + \left(1 - x_1\right)^2$$

is minimized.

```
      USE UMINF_INT
      USE U4INF_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=2)
!
      INTEGER    IPARAM(7), L, NOUT
      REAL       F, RPARAM(7), X(N), XGUESS(N), &
                XSCALE(N)
      EXTERNAL   ROSBRK
!
      DATA XGUESS/-1.2E0, 1.0E0/
!
!                                  Relax gradient tolerance stopping
!                                  criterion
      CALL U4INF (IPARAM, RPARAM)
      RPARAM(1) = 10.0E0*RPARAM(1)
!                                  Minimize Rosenbrock function using
!                                  initial guesses of -1.2 and 1.0
      CALL UMINF (ROSBRK, X, XGUESS=XGUESS, IPARAM=IPARAM, RPARAM=RPARAM, &
      FVALUE=F)
!                                  Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) X, F, (IPARAM(L),L=3,5)
!
99999 FORMAT (’  The solution is ’, 6X, 2F8.3, //, ’  The function ’, &
            ’value is ’, F8.3, //, ’  The number of iterations is ’, &
            10X, I3, /, ’  The number of function evaluations is ’, &
            I3, /, ’  The number of gradient evaluations is ’, I3)
!
      END
!
      SUBROUTINE ROSBRK (N, X, F)
      INTEGER    N
      REAL       X(N), F
!
      F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
!
      RETURN
      END
```

Output

```
The solution is          1.000   1.000

The function value is    0.000

The number of iterations is           15
The number of function evaluations is  40
The number of gradient evaluations is  19
```

Comments

1. Workspace may be explicitly provided, if desired, by use of U2INF/DU2INF. The reference is:

   ```
   CALL U2INF (FCN, N, XGUESS, XSCALE, FSCALE, IPARAM,
   RPARAM, X,FVALUE, WK)
   ```

   The additional argument is:

   WK — Work vector of length N(N + 8). WK contains the following information on output: The second N locations contain the last step taken. The third N locations contain the last Newton step. The fourth N locations contain an estimate of the gradient at the solution. The final N^2 locations contain the Cholesky factorization of a BFGS approximation to the Hessian at the solution.

2. Informational errors

   Type   Code

3	1	Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.
4	2	The iterates appear to be converging to a noncritical point.
4	3	Maximum number of iterations exceeded.
4	4	Maximum number of function evaluations exceeded.
4	5	Maximum number of gradient evaluations exceeded.
4	6	Five consecutive steps have been taken with the maximum step length.
2	7	Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or STEPTL is too big.
3	8	The last global step failed to locate a lower point than the current X value.

3. The first stopping criterion for UMINF occurs when the infinity norm of the scaled gradient is less than the given gradient tolerance (RPARAM(1)). The second stopping criterion for UMINF occurs when the scaled distance between the last two steps is less than the step tolerance (RPARAM(2)).

4. If the default parameters are desired for UMINF, then set IPARAM(1) to zero and call the routine UMINF. Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM, then the following steps should be taken before calling UMINF:

CALL U4INF (IPARAM, RPARAM)

Set nondefault values for desired IPARAM, RPARAM elements.

Note that the call to U4INF will set IPARAM and RPARAM to their default values so only nondefault values need to be set above.

The following is a list of the parameters and the default values:

IPARAM — Integer vector of length 7.

IPARAM(1) = Initialization flag.

IPARAM(2) = Number of good digits in the function.

Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.

Default: 100.

IPARAM(4) = Maximum number of function evaluations.

Default: 400.

IPARAM(5) = Maximum number of gradient evaluations.

Default: 400.

IPARAM(6) = Hessian initialization parameter.

If IPARAM(6) = 0, the Hessian is initialized to the identity matrix; otherwise, it is initialized to a diagonal matrix containing

$$\max\left(\left|f\left(t\right)\right|, f_s\right) * s_i^2$$

on the diagonal where t = XGUESS, f_S = FSCALE, and s = XSCALE.

Default: 0.

IPARAM(7) = Maximum number of Hessian evaluations.

Default: Not used in UMINF.

RPARAM — Real vector of length 7.

RPARAM(1) = Scaled gradient tolerance.

The i-th component of the scaled gradient at
x is calculated as

$$\frac{\left|g_i\right| * \max\left(\left|x_i\right|, 1/s_i\right)}{\max\left(\left|f\left(x\right)\right|, f_s\right)}$$

where $g = \nabla f(x)$, s = XSCALE, and f_S = FSCALE.

Default:

$$\sqrt{\varepsilon}, \sqrt[3]{\varepsilon}$$

in double where ε is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)

The i-th component of the scaled step between two points x and y is computed as

$$\frac{\left|x_i - y_i\right|}{\max\left(\left|x_i\right|, 1/s_i\right)}$$

where s = XSCALE.

Default: ε2/3 where ε is the machine precision.

RPARAM(3) = Relative function tolerance.

Default: $\max(10^{-10}, \varepsilon^{2/3})$, $\max(10^{-20}, \varepsilon^{2/3})$ in double where ε is the machine precision.

RPARAM(4) = Absolute function tolerance.

Default: Not used in UMINF.

RPARAM(5) = False convergence tolerance.

Default: Not used in UMINF.

RPARAM(6) = Maximum allowable step size.
Default: 1000 max(ε_1, ε_2) where

$$\varepsilon_1 = \sqrt{\sum_{i=1}^{n} \left(s_i t_i\right)^2}, \; \varepsilon_2 = \left\|s\right\|_2, \; s = \text{XSCALE, and } t = \text{XGUESS}$$

RPARAM(7) = Size of initial trust region radius.
Default: Not used in UMINF.

If double precision is required, then DU4INF is called, and RPARAM is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to “Error Handling” in the Introduction.

Description

The routine UMINF uses a quasi-Newton method to find the minimum of a function $f(x)$ of n variables. Only function values are required. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} f\left(x\right)$$

Given a starting point x_c, the search direction is computed according to the formula

$$d = -B^{-1} g_c$$

where B is a positive definite approximation of the Hessian and g_c is the gradient evaluated at x_c. A line search is then used to find a new point

$$x_n = x_c + \lambda d, \lambda > 0$$

such that

$$f(x_n) \le f(x_c) + \alpha g^T d, \quad \alpha \in (0, 0.5)$$

Finally, the optimality condition $\|g(x)\| = \varepsilon$ is checked where ε is a gradient tolerance.

When optimality is not achieved, B is updated according to the BFGS formula

$$B \leftarrow B - \frac{Bss^T B}{s^T Bs} + \frac{yy^T}{y^T s}$$

where $s = x_n - x_c$ and $y = g_n - g_c$. Another search direction is then computed to begin the next iteration. For more details, see Dennis and Schnabel (1983, Appendix A).

Since a finite-difference method is used to estimate the gradient, for some single precision calculations, an inaccurate estimate of the gradient may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended. Also, whenever the exact gradient can be easily provided, IMSL routine UMING (page 1202) should be used instead.

UMING

Minimizes a function of N variables using a quasi-Newton method and a user-supplied gradient.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function to be minimized. The usage is CALL FCN (N, X, F), where

N – Length of X.   (Input)

X – Vector of length N at which point the function is evaluated.   (Input)
X should not be changed by FCN.

F – The computed function value at the point X.   (Output)

FCN must be declared EXTERNAL in the calling program.

GRAD — User-supplied SUBROUTINE to compute the gradient at the point X. The usage is CALL GRAD (N, X, G), where

N – Length of X and G.   (Input)
X – Vector of length N at which point the function is evaluated.   (Input)
X should not be changed by GRAD .
G – The gradient evaluated at the point X.   (Output)

GRAD must be declared EXTERNAL in the calling program.

X — Vector of length N containing the computed solution.   (Output)

Optional Arguments

N — Dimension of the problem.   (Input)
Default: N = size (X,1).

XGUESS — Vector of length N containing the initial guess of the minimum.   (Input)
Default: XGUESS = 0.0.

XSCALE — Vector of length N containing the diagonal scaling matrix for the variables. (Input)
XSCALE is used mainly in scaling the gradient and the distance between two points. In the absence of other information, set all entries to 1.0.
Default: XSCALE = 1.0.

FSCALE — Scalar containing the function scaling.   (Input)
FSCALE is used mainly in scaling the gradient. In the absence of other information, set

FSCALE to 1.0.
Default: FSCALE = 1.0.

IPARAM — Parameter vector of length 7. (Input/Output)
Set IPARAM(1) to zero for default values of IPARAM and RPARAM. See Comment 4.
Default: IPARAM = 0.

RPARAM — Parameter vector of length 7. (Input/Output)
See Comment 4.

FVALUE — Scalar containing the value of the function at the computed solution. (Output)

FORTRAN 90 Interface

Generic: CALL UMING (FCN, GRAD, X [,…])

Specific: The specific interface names are S_UMING and D_UMING.

FORTRAN 77 Interface

Single: CALL UMING (FCN, GRAD, N, XGUESS, XSCALE, FSCALE, IPARAM, RPARAM, X, FVALUE)

Double: The double precision name is DUMING.

Example

The function

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + \left(1 - x_1\right)^2$$

is minimized. Default values for parameters are used.

```
      USE UMING_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=2)
!
      INTEGER    IPARAM(7), L, NOUT
      REAL       F, X(N), XGUESS(N)
      EXTERNAL   ROSBRK, ROSGRD
!
      DATA XGUESS/-1.2E0, 1.0E0/
!
      IPARAM(1) = 0
!                                 Minimize Rosenbrock function using
!                                 initial guesses of -1.2 and 1.0
      CALL UMING (ROSBRK, ROSGRD, X, XGUESS=XGUESS, IPARAM=IPARAM, FVALUE=F)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) X, F, (IPARAM(L),L=3,5)
```

```
!
99999 FORMAT (’  The solution is ’, 6X, 2F8.3, //, ’  The function ’, &
            ’value is ’, F8.3, //, ’  The number of iterations is ’, &
            10X, I3, /, ’  The number of function evaluations is ’, &
            I3, /, ’  The number of gradient evaluations is ’, I3)
!
      END
!
      SUBROUTINE ROSBRK (N, X, F)
      INTEGER    N
      REAL       X(N), F
!
      F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
!
      RETURN
      END
!
      SUBROUTINE ROSGRD (N, X, G)
      INTEGER    N
      REAL       X(N), G(N)
!
      G(1) = -4.0E2*(X(2)-X(1)*X(1))*X(1) - 2.0E0*(1.0E0-X(1))
      G(2) = 2.0E2*(X(2)-X(1)*X(1))
!
      RETURN
      END
```

Output

```
The solution is          1.000   1.000

The function value is    0.000

The number of iterations is            18
The number of function evaluations is  31
The number of gradient evaluations is  22
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `U2ING/DU2ING`. The reference is:

   `CALL U2ING (FCN, GRAD, N, XGUESS, XSCALE, FSCALE, IPARAM, RPARAM, X, FVALUE, WK)`

   The additional argument is

   WK — Work vector of length `N * (N + 8)`. `WK` contains the following information on output: The second `N` locations contain the last step taken. The third `N` locations contain the last Newton step. The fourth `N` locations contain an estimate of the gradient at the solution. The final N^2 locations contain the Cholesky factorization of a `BFGS` approximation to the Hessian at the solution.

2. Informational errors

Type	Code	
3	1	Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.
4	2	The iterates appear to be converging to a noncritical point.
4	3	Maximum number of iterations exceeded.
4	4	Maximum number of function evaluations exceeded.
4	5	Maximum number of gradient evaluations exceeded.
4	6	Five consecutive steps have been taken with the maximum step length.
2	7	Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or STEPTL is too big.
3	8	The last global step failed to locate a lower point than the current X value.

3. The first stopping criterion for UMING occurs when the infinity norm of the scaled gradient is less than the given gradient tolerance (RPARAM(1)). The second stopping criterion for UMING occurs when the scaled distance between the last two steps is less than the step tolerance (RPARAM(2)).

4. If the default parameters are desired for UMING, then set IPARAM(1) to zero and call routine UMING (page 1202). Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM, then the following steps should be taken before calling UMING:

   CALL U4INF (IPARAM, RPARAM)
   Set nondefault values for desired IPARAM, RPARAM elements.

   Note that the call to U4INF will set IPARAM and RPARAM to their default values so only nondefault values need to be set above.

The following is a list of the parameters and the default values:

IPARAM — Integer vector of length 7.

IPARAM(1) = Initialization flag.

IPARAM(2) = Number of good digits in the function.
Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.
Default: 100.

IPARAM(4) = Maximum number of function evaluations.
Default: 400.

IPARAM(5) = Maximum number of gradient evaluations.
Default: 400.

IPARAM(6) = Hessian initialization parameter

If IPARAM(6) = 0, the Hessian is initialized to the identity matrix; otherwise, it is initialized to a diagonal matrix containing

$$\max\left(\left|f\left(t\right)\right|, f_s\right) * s_i^2$$

on the diagonal where t = XGUESS, f_s = FSCALE, and s = XSCALE.

Default: 0.

IPARAM(7) = Maximum number of Hessian evaluations.

Default: Not used in UMING.

RPARAM — Real vector of length 7.

RPARAM(1) = Scaled gradient tolerance.

The i-th component of the scaled gradient at
x is calculated as

$$\frac{\left|g_i\right| * \max\left(\left|x_i\right|, 1/s_i\right)}{\max\left(\left|f\left(x\right)\right|, f_s\right)}$$

where $g = \nabla f(x)$, s = XSCALE, and f_s = FSCALE.

Default:

$$\sqrt{\varepsilon}, \sqrt[3]{\varepsilon}$$

in double where ε is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)

The i-th component of the scaled step between two points x and y is computed as

$$\frac{\left|x_i - y_i\right|}{\max\left(\left|x_i\right|, 1/s_i\right)}$$

where s = XSCALE.

Default: $\varepsilon^{2/3}$ where ε is the machine precision.

RPARAM(3) = Relative function tolerance.

Default: $\max(10^{-10}, \varepsilon^{2/3})$, $\max(10^{-20}, \varepsilon^{2/3})$ in double where ε is the machine precision.

RPARAM(4) = Absolute function tolerance.

Default: Not used in UMING.

`RPARAM`(5) = False convergence tolerance.
Default: Not used in `UMING`.

`RPARAM`(6) = Maximum allowable step size.
Default: 1000 max(ε_1, ε_2) where

$$\varepsilon_1 = \sqrt{\sum_{i=1}^{n} (s_i t_i)^2}$$

$\varepsilon_2 = \| s \|_2$, s = `XSCALE`, and t = `XGUESS`.

`RPARAM`(7) = Size of initial trust region radius.
Default: Not used in `UMING`.

If double precision is required, then `DU4INF` is called, and `RPARAM` is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to “Error Handling” in the Introduction.

Description

The routine `UMING` uses a quasi-Newton method to find the minimum of a function $f(x)$ of n variables. Function values and first derivatives are required. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

Given a starting point x_c, the search direction is computed according to the formula

$$d = -B^{-1} g_c$$

where B is a positive definite approximation of the Hessian and g_c is the gradient evaluated at x_c. A line search is then used to find a new point

$$x_n = x_c + \lambda d, \lambda > 0$$

such that

$$f(x_n) \le f(x_c) + \alpha g^T d, \quad \alpha \in (0, 0.5)$$

Finally, the optimality condition $\|g(x)\| = \varepsilon$ is checked where ε is a gradient tolerance.

When optimality is not achieved, B is updated according to the BFGS formula

$$B \leftarrow B - \frac{Bss^T B}{s^T Bs} + \frac{yy^T}{y^T s}$$

where $s = x_n - x_c$ and $y = g_n - g_c$. Another search direction is then computed to begin the next iteration. For more details, see Dennis and Schnabel (1983, Appendix A).

UMIDH

Minimizes a function of N variables using a modified Newton method and a finite-difference Hessian.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function to be minimized. The usage is CALL FCN (N, X, F), where

N – Length of X. (Input)

X – Vector of length N at which point the function is evaluated. (Input) X should not be changed by FCN.

F – The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

GRAD — User-supplied SUBROUTINE to compute the gradient at the point X. The usage is CALL GRAD (N, X, G), where

N – Length of X and G. (Input)

X – The point at which the gradient is evaluated. (Input) X should not be changed by GRAD.

G – The gradient evaluated at the point X. (Output)

GRAD must be declared EXTERNAL in the calling program.

X — Vector of length N containing the computed solution. (Output)

Optional Arguments

N — Dimension of the problem. (Input)
Default: N = size (X,1).

XGUESS — Vector of length N containing initial guess. (Input)
Default: XGUESS = 0.0.

XSCALE — Vector of length N containing the diagonal scaling matrix for the variables. (Input)
XSCALE is used mainly in scaling the gradient and the distance between two points. In the absence of other information, set all entries to 1.0.
Default: XSCALE = 1.0.

FSCALE — Scalar containing the function scaling. (Input)
FSCALE is used mainly in scaling the gradient. In the absence of other information, set FSCALE to 1.0.
Default: FSCALE = 1.0.

IPARAM — Parameter vector of length 7. (Input/Output)
Set IPARAM(1) to zero for default values of IPARAM and RPARAM. See Comment 4.
Default: IPARAM = 0.

RPARAM — Parameter vector of length 7. (Input/Output)
See Comment 4.

FVALUE — Scalar containing the value of the function at the computed solution. (Output)

FORTRAN 90 Interface

Generic: CALL UMIDH (FCN, GRAD, X [,…])

Specific: The specific interface names are S_UMIDH and D_UMIDH.

FORTRAN 77 Interface

Single: CALL UMIDH (FCN, GRAD, N, XGUESS, XSCALE, FSCALE, IPARAM, RPARAM, X, FVALUE)

Double: The double precision name is DUMIDH.

Example

The function

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + \left(1 - x_1\right)^2$$

is minimized. Default values for parameters are used.

```
      USE UMIDH_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=2)
!
      INTEGER    IPARAM(7), L, NOUT
      REAL       F, X(N), XGUESS(N)
      EXTERNAL   ROSBRK, ROSGRD
!
      DATA XGUESS/-1.2E0, 1.0E0/
!
      IPARAM(1) = 0
!                                 Minimize Rosenbrock function using
!                                 initial guesses of -1.2 and 1.0
      CALL UMIDH (ROSBRK, ROSGRD, X, XGUESS=XGUESS, IPARAM=IPARAM, FVALUE=F)
!                                 Print results
```

```
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) X, F, (IPARAM(L),L=3,5), IPARAM(7)
!
99999 FORMAT ('  The solution is ', 6X, 2F8.3, //, '  The function ', &
             'value is ', F8.3, //, '  The number of iterations is ', &
             10X, I3, /, '  The number of function evaluations is ', &
             I3, /, '  The number of gradient evaluations is ', I3, /, &
             '  The number of Hessian evaluations is  ', I3)
!
      END
!
      SUBROUTINE ROSBRK (N, X, F)
      INTEGER    N
      REAL       X(N), F
!
      F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
!
      RETURN
      END
!
      SUBROUTINE ROSGRD (N, X, G)
      INTEGER    N
      REAL       X(N), G(N)
!
      G(1) = -4.0E2*(X(2)-X(1)*X(1))*X(1) - 2.0E0*(1.0E0-X(1))
      G(2) = 2.0E2*(X(2)-X(1)*X(1))
!
      RETURN
      END
```

Output

```
The solution is          1.000   1.000

The function value is    0.000

The number of iterations is            21
The number of function evaluations is  30
The number of gradient evaluations is  22
The number of Hessian evaluations is   21
```

Comments

1. Workspace may be explicitly provided, if desired, by use of U2IDH/DU2IDH. The reference is:

   1CALL U2IDH (FCN, GRAD, N, XGUESS, XSCALE, FSCALE, IPARAM, RPARAM, X, FVALUE, WK)

   The additional argument is:

   WK — Work vector of length N * (N + 9). WK contains the following information on output: The second N locations contain the last step taken. The third N locations contain the last Newton step. The fourth N locations contain an estimate of the gradient at the solution. The final N^2 locations contain the Hessian at the approximate solution.

2. Informational errors

Type	Code	
3	1	Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.
4	2	The iterates appear to be converging to a noncritical point.
4	3	Maximum number of iterations exceeded.
4	4	Maximum number of function evaluations exceeded.
4	5	Maximum number of gradient evaluations exceeded.
4	6	Five consecutive steps have been taken with the maximum step length.
2	7	Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or STEPTL is too big.
4	7	Maximum number of Hessian evaluations exceeded.
3	8	The last global step failed to locate a lower point than the current X value.

3. The first stopping criterion for UMIDH occurs when the norm of the gradient is less than the given gradient tolerance (RPARAM(1)). The second stopping criterion for UMIDH occurs when the scaled distance between the last two steps is less than the step tolerance (RPARAM(2)).

4. If the default parameters are desired for UMIDH, then set IPARAM(1) to zero and call routine UMIDH. Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM, then the following steps should be taken before calling UMIDH:

   CALL U4INF (IPARAM, RPARAM)

   Set nondefault values for desired IPARAM, RPARAM elements.

   Note that the call to U4INF will set IPARAM and RPARAM to their default values so only nondefault values need to be set above.

   The following is a list of the parameters and the default values:

   IPARAM — Integer vector of length 7.
   IPARAM(1) = Initialization flag.

   IPARAM(2) = Number of good digits in the function.
   Default: Machine dependent.

   IPARAM(3) = Maximum number of iterations.
   Default: 100.

IPARAM(4) = Maximum number of function evaluations.
Default: 400.

IPARAM(5) = Maximum number of gradient evaluations.
Default: 400.

IPARAM(6) = Hessian initialization parameter
Default: Not used in UMIDH.

IPARAM(7) = Maximum number of Hessian evaluations.
Default:100

RPARAM — Real vector of length 7.

RPARAM(1) = Scaled gradient tolerance.
The *i*-th component of the scaled gradient at *x* is calculated as

$$\frac{|g_i| * \max\left(|x_i|, 1/s_i\right)}{\max\left(\left|f\left(x\right)\right|, f_s\right)}$$

where $g = \nabla f(x)$, s = XSCALE, and f_s = FSCALE.
Default:

$$\sqrt{\varepsilon}, \sqrt[3]{\varepsilon}$$

in double where ε is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)

The *i*-th component of the scaled step between two points *x* and *y* is computed as

$$\frac{|x_i - y_i|}{\max\left(|x_i|, 1/s_i\right)}$$

where s = XSCALE.
Default: $\varepsilon^{2/3}$ where ε is the machine precision.

RPARAM(3) = Relative function tolerance.

Default: $\max(10^{-10}, \varepsilon^{2/3})$, $\max(10^{-20}, \varepsilon^{2/3})$ in double where ε is the machine precision.

RPARAM(4) = Absolute function tolerance.

Default: Not used in UMIDH.

RPARAM(5) = False convergence tolerance.

Default: 100ε where ε is the machine precision.

RPARAM(6) = Maximum allowable step size.

Default: 1000 max(ε_1, ε_2) where

$$\varepsilon_1 = \sqrt{\sum_{i=1}^{n} \left(s_i t_i\right)^2}$$

$\varepsilon_2 = \| s \|_2$, s = XSCALE, and t = XGUESS.

RPARAM(7) = Size of initial trust region radius.

Default: Based on initial scaled Cauchy step.

If double precision is required, then DU4INF is called, and RPARAM is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to “Error Handling” in the Introduction.

Description

The routine UMIDH uses a modified Newton method to find the minimum of a function $f(x)$ of n variables. First derivatives must be provided by the user. The algorithm computes an optimal locally constrained step (Gay 1981) with a trust region restriction on the step. It handles the case that the Hessian is indefinite and provides a way to deal with negative curvature. For more details, see Dennis and Schnabel (1983, Appendix A) and Gay (1983).

Since a finite-difference method is used to estimate the Hessian for some single precision calculations, an inaccurate estimate of the Hessian may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended. Also, whenever the exact Hessian can be easily provided, IMSL routine UMIAH (page 1213) should be used instead.

UMIAH

Minimizes a function of N variables using a modified Newton method and a user-supplied Hessian.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function to be minimized. The usage is CALL FCN (N, X, F), where

N – Length of X. (Input)

X – Vector of length N at which point the function is evaluated. (Input)
X should not be changed by FCN.

F – The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

GRAD — User-supplied SUBROUTINE to compute the gradient at the point X. The usage is CALL GRAD (N, X, G), where

N – Length of X and G. (Input)

X – Vector of length N at which point the gradient is evaluated. (Input)
X should not be changed by GRAD.

G – The gradient evaluated at the point X. (Output)

GRAD must be declared EXTERNAL in the calling program.

HESS — User-supplied SUBROUTINE to compute the Hessian at the point X. The usage is CALL HESS (N, X, H, LDH), where

N – Length of X. (Input)

X – Vector of length N at which point the Hessian is evaluated. (Input)
X should not be changed by HESS.

H – The Hessian evaluated at the point X. (Output)

LDH – Leading dimension of H exactly as specified in the dimension statement of the calling program. LDH must be equal to N in this routine. (Input)

HESS must be declared EXTERNAL in the calling program.

X — Vector of length N containing the computed solution. (Output)

Optional Arguments

N — Dimension of the problem. (Input)
Default: N = size (X,1).

XGUESS — Vector of length N containing initial guess. (Input)
Default: XGUESS = 0.0.

XSCALE — Vector of length N containing the diagonal scaling matrix for the variables. (Input)
XSCALE is used mainly in scaling the gradient and the distance between two points. In

the absence of other information, set all entries to 1.0.
Default: XSCALE = 1.0.

FSCALE — Scalar containing the function scaling. (Input)
FSCALE is used mainly in scaling the gradient. In the absence of other information, set FSCALE to 1.0.
Default: FSCALE = 1.0.

IPARAM — Parameter vector of length 7. (Input/Output)
Set IPARAM(1) to zero for default values of IPARAM and RPARAM. See Comment 4.
Default: IPARAM = 0.

RPARAM — Parameter vector of length 7. (Input/Output)
See Comment 4.

FVALUE — Scalar containing the value of the function at the computed solution. (Output)

FORTRAN 90 Interface

Generic: CALL UMIAH (FCN, GRAD, HESS, X, [,…])

Specific: The specific interface names are S_UMIAH and D_UMIAH.

FORTRAN 77 Interface

Single:
```
CALL UMIAH (FCN, GRAD, HESS, N, XGUESS, XSCALE, FSCALE,
IPARAM, RPARAM, X, FVALUE)
```

Double: The double precision name is DUMIAH.

Example

The function

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + \left(1 - x_1\right)^2$$

is minimized. Default values for parameters are used.

```
      USE UMIAH_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=2)
!
      INTEGER    IPARAM(7), L, NOUT
      REAL       F, FSCALE, RPARAM(7), X(N), &
                XGUESS(N), XSCALE(N)
      EXTERNAL   ROSBRK, ROSGRD, ROSHES
!
      DATA XGUESS/-1.2E0, 1.0E0/, XSCALE/1.0E0, 1.0E0/, FSCALE/1.0E0/
!
```

```
      IPARAM(1) = 0
!                                 Minimize Rosenbrock function using
!                                 initial guesses of -1.2 and 1.0
       CALL UMIAH (ROSBRK, ROSGRD, ROSHES, X, XGUESS=XGUESS, IPARAM=IPARAM, &
                   FVALUE=F)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) X, F, (IPARAM(L),L=3,5), IPARAM(7)
!
99999 FORMAT (’  The solution is ’, 6X, 2F8.3, //, ’  The function ’, &
            ’value is ’, F8.3, //, ’  The number of iterations is ’, &
            10X, I3, /, ’  The number of function evaluations is ’, &
            I3, /, ’  The number of gradient evaluations is ’, I3, /, &
            ’  The number of Hessian evaluations is  ’, I3)
!
      END
!
      SUBROUTINE ROSBRK (N, X, F)
      INTEGER    N
      REAL       X(N), F
!
      F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
!
      RETURN
      END
!
      SUBROUTINE ROSGRD (N, X, G)
      INTEGER    N
      REAL       X(N), G(N)
!
      G(1) = -4.0E2*(X(2)-X(1)*X(1))*X(1) - 2.0E0*(1.0E0-X(1))
      G(2) = 2.0E2*(X(2)-X(1)*X(1))
!
      RETURN
      END
!
      SUBROUTINE ROSHES (N, X, H, LDH)
      INTEGER    N, LDH
      REAL       X(N), H(LDH,N)
!
      H(1,1) = -4.0E2*X(2) + 1.2E3*X(1)*X(1) + 2.0E0
      H(2,1) = -4.0E2*X(1)
      H(1,2) = H(2,1)
      H(2,2) = 2.0E2
!
      RETURN
      END
```

Output

```
The solution is          1.000   1.000

The function value is    0.000

The number of iterations is            21
The number of function evaluations is  31
The number of gradient evaluations is  22
The number of Hessian evaluations is   21
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `U2IAH/DU2IAH`. The reference is:

   ```
   CALL U2IAH (FCN, GRAD, HESS, N, XGUESS, XSCALE, FSCALE, IPARAM,
   RPARAM, X, FVALUE, WK)
   ```

   The additional argument is:

   WK — Work vector of length `N * (N + 9)`. `WK` contains the following information on output: The second `N` locations contain the last step taken. The third `N` locations contain the last Newton step. The fourth `N` locations contain an estimate of the gradient at the solution. The final N^2 locations contain the Hessian at the approximate solution.

2. Informational errors

Type	Code	
3	1	Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.
4	2	The iterates appear to be converging to a noncritical point.
4	3	Maximum number of iterations exceeded.
4	4	Maximum number of function evaluations exceeded.
4	5	Maximum number of gradient evaluations exceeded.
4	6	Five consecutive steps have been taken with the maximum step length.
2	7	Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or `STEPTL` is too big.
4	7	Maximum number of Hessian evaluations exceeded.
3	8	The last global step failed to locate a lower point than the current `X` value.

3. The first stopping criterion for `UMIAH` occurs when the norm of the gradient is less than the given gradient tolerance (`RPARAM`(1)). The second stopping criterion for `UMIAH` occurs when the scaled distance between the last two steps is less than the step tolerance (`RPARAM`(2)).

4. If the default parameters are desired for `UMIAH`, then set `IPARAM`(1) to zero and call the routine `UMIAH`. Otherwise, if any nondefault parameters are desired for `IPARAM` or `RPARAM`, then the following steps should be taken before calling `UMIAH`:

   `CALL U4INF` (`IPARAM`, `RPARAM`)

   Set nondefault values for desired `IPARAM`, `RPARAM` elements.

Note that the call to U4INF will set IPARAM and RPARAM to their default values so only nondefault values need to be set above.

The following is a list of the parameters and the default values:

IPARAM — Integer vector of length 7.
IPARAM(1) = Initialization flag.

IPARAM(2) = Number of good digits in the function.
Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.
Default: 100.

IPARAM(4) = Maximum number of function evaluations.
Default: 400.

IPARAM(5) = Maximum number of gradient evaluations.
Default: 400.

IPARAM(6) = Hessian initialization parameter
Default: Not used in UMIAH.

IPARAM(7) = Maximum number of Hessian evaluations.
Default: 100.

RPARAM — Real vector of length 7.
RPARAM(1) = Scaled gradient tolerance.
The *i*-th component of the scaled gradient at *x* is calculated as

$$\frac{|g_i| * \max\left(|x_i|, 1/s_i\right)}{\max\left(\left|f\left(x\right)\right|, f_s\right)}$$

where $g = \nabla f(x)$, s = XSCALE, and f_s = FSCALE.
Default:

$$\sqrt{\varepsilon}, \sqrt[3]{\varepsilon}$$

in double where ε is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)
The *i*-th component of the scaled step between two points *x* and *y* is computed as

$$\frac{|x_i - y_i|}{\max\left(|x_i|, 1/s_i\right)}$$

where s = XSCALE.
Default: $\varepsilon^{2/3}$ where ε is the machine precision.

RPARAM(3) = Relative function tolerance.
Default: max(10^{-10}, $\varepsilon^{2/3}$), max(10^{-20}, $\varepsilon^{2/3}$) in double where ε is the machine precision.

RPARAM(4) = Absolute function tolerance.
Default: Not used in UMIAH.

RPARAM(5) = False convergence tolerance.
Default: 100ε where ε is the machine precision.

RPARAM(6) = Maximum allowable step size.
Default: 1000 max(ε_1, ε_2) where

$$\varepsilon_1 = \sqrt{\sum_{i=1}^{n} \left(s_i t_i\right)^2}$$

$\varepsilon_2 = \| s \|_2$, s = XSCALE, and t = XGUESS.

RPARAM(7) = Size of initial trust region radius.
Default: based on the initial scaled Cauchy step.

If double precision is required, then DU4INF is called, and RPARAM is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to “Error Handling” in the Introduction.

Description

The routine UMIAH uses a modified Newton method to find the minimum of a function $f(x)$ of n variables. First and second derivatives must be provided by the user. The algorithm computes an optimal locally constrained step (Gay 1981) with a trust region restriction on the step. This algorithm handles the case where the Hessian is indefinite and provides a way to deal with negative curvature. For more details, see Dennis and Schnabel (1983, Appendix A) and Gay (1983).

UMCGF

Minimizes a function of N variables using a conjugate gradient algorithm and a finite-difference gradient.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function to be minimized. The usage is CALL FCN (N, X, F), where

N – Length of X. (Input)

X – The point at which the function is evaluated. (Input)
X should not be changed by FCN.

F – The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

DFPRED — A rough estimate of the expected reduction in the function. (Input)
DFPRED is used to determine the size of the initial change to X.

X — Vector of length N containing the computed solution. (Output)

Optional Arguments

N — Dimension of the problem. (Input)
Default: N = size (X,1).

XGUESS — Vector of length N containing the initial guess of the minimum. (Input)
Default: XGUESS = 0.0.

XSCALE — Vector of length N containing the diagonal scaling matrix for the variables. (Input)
Default: XSCALE = 1.0.

GRADTL — Convergence criterion. (Input)
The calculation ends when the sum of squares of the components of G is less than GRADTL.
Default: GRADTL = 1.e-4.

MAXFN — Maximum number of function evaluations. (Input)
If MAXFN is set to zero, then no restriction on the number of function evaluations is set.
Default: MAXFN = 0.

G — Vector of length N containing the components of the gradient at the final parameter estimates. (Output)

FVALUE — Scalar containing the value of the function at the computed solution. (Output)

FORTRAN 90 Interface

Generic: CALL UMCGF (FCN, DFPRED, X [,…])

Specific: The specific interface names are `S_UMCGF` and `D_UMCGF`.

FORTRAN 77 Interface

Single: `CALL UMCGF (FCN, N, XGUESS, XSCALE, GRADTL, MAXFN, DFPRED, X, G, FVALUE)`

Double: The double precision name is `DUMCGF`.

Example

The function

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + \left(1 - x_1\right)^2$$

is minimized and the solution is printed.

```
      USE UMCGF_INT
      USE UMACH_INT
!                                 Declaration of variables
      INTEGER    N
      PARAMETER  (N=2)
!
      INTEGER    I, MAXFN, NOUT
      REAL       DFPRED, FVALUE, G(N), GRADTL, X(N), XGUESS(N)
      EXTERNAL   ROSBRK
!
      DATA XGUESS/-1.2E0, 1.0E0/
!
      DFPRED = 0.2
      GRADTL = 1.0E-6
      MAXFN  = 100
!                                 Minimize the Rosenbrock function
      CALL UMCGF (ROSBRK, DFPRED, X, XGUESS=XGUESS, GRADTL=GRADTL, &
                 G=G, FVALUE=FVALUE)
!                                 Print the results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) (X(I),I=1,N), FVALUE, (G(I),I=1,N)
99999 FORMAT (’  The solution is ’, 2F8.3, //, ’  The function ’, &
            ’evaluated at the solution is ’, F8.3, //, ’  The ’, &
            ’gradient is ’, 2F8.3, /)
!
      END
!
      SUBROUTINE ROSBRK (N, X, F)
      INTEGER    N
      REAL       X(N), F
!
      F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
      RETURN
      END
```

Output

```
The solution is    0.999   0.998

The function evaluated at the solution is    0.000

The gradient is   -0.001   0.000
```

Comments

1. Workspace may be explicitly provided, if desired, by use of U2CGF/DU2CGF. The reference is:

   ```
   CALL U2CGF (FCN, N, XGUESS, XSCALE, GRADTL, MAXFN, DFPRED, X, G,
   FVALUE, S, RSS, RSG, GINIT, XOPT, GOPT)
   ```

   The additional arguments are as follows:

   S — Vector of length N used for the search direction in each iteration.

   RSS — Vector of length N containing conjugacy information.

   RSG — Vector of length N containing conjugacy information.

   GINIT — Vector of length N containing the gradient values at the start of an iteration.

   XOPT — Vector of length N containing the parameter values that yield the least calculated value for FVALUE.

   GOPT — Vector of length N containing the gradient values that yield the least calculated value for FVALUE.

2. Informational errors

Type	Code	
4	1	The line search of an integration was abandoned. This error may be caused by an error in gradient.
4	2	The calculation cannot continue because the search is uphill.
4	3	The iteration was terminated because MAXFN was exceeded.
3	4	The calculation was terminated because two consecutive iterations failed to reduce the function.

3. Because of the close relation between the conjugate-gradient method and the method of steepest descent, it is very helpful to choose the scale of the variables in a way that balances the magnitudes of the components of a typical gradient vector. It can be particularly inefficient if a few components of the gradient are much larger than the rest.

4. If the value of the parameter GRADTL in the argument list of the routine is set to zero, then the subroutine will continue its calculation until it stops reducing the objective function. In this case, the usual behavior is that changes in the objective function become dominated by computer rounding errors before precision is lost in the gradient

vector. Therefore, because the point of view has been taken that the user requires the least possible value of the function, a value of the objective function that is small due to computer rounding errors can prevent further progress. Hence, the precision in the final values of the variables may be only about half the number of significant digits in the computer arithmetic, but the least value of `FVALUE` is usually found to be quite accurate.

Description

The routine `UMCGF` uses a conjugate gradient method to find the minimum of a function $f(x)$ of n variables. Only function values are required.

The routine is based on the version of the conjugate gradient algorithm described in Powell (1977). The main advantage of the conjugate gradient technique is that it provides a fast rate of convergence without the storage of any matrices. Therefore, it is particularly suitable for unconstrained minimization calculations where the number of variables is so large that matrices of dimension n cannot be stored in the main memory of the computer. For smaller problems, however, a routine such as routine `UMINF` (page 1196), is usually more efficient because each iteration makes use of additional information from previous iterations.

Since a finite-difference method is used to estimate the gradient for some single precision calculations, an inaccurate estimate of the gradient may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended. Also, whenever the exact gradient can be easily provided, routine `UMCGG` (page 1223) should be used instead.

UMCGG

Minimizes a function of `N` variables using a conjugate gradient algorithm and a user-supplied gradient.

Required Arguments

FCN — User-supplied `SUBROUTINE` to evaluate the function to be minimized. The usage is `CALL FCN (N, X, F)`, where

`N` – Length of `X`. (Input)

`X` – The point at which the function is evaluated. (Input)
`X` should not be changed by `FCN`.

`F` – The computed function value at the point `X`. (Output)

`FCN` must be declared `EXTERNAL` in the calling program.

GRAD — User-supplied `SUBROUTINE` to compute the gradient at the point `X`. The usage is `CALL GRAD (N, X, G)`, where

`N` – Length of `X` and `G`. (Input)

X – The point at which the gradient is evaluated. (Input)
X should not be changed by GRAD.

G – The gradient evaluated at the point X. (Output)

GRAD must be declared EXTERNAL in the calling program.

DFPRED — A rough estimate of the expected reduction in the function. (Input) DFPRED is used to determine the size of the initial change to X.

X — Vector of length N containing the computed solution. (Output)

Optional Arguments

N — Dimension of the problem. (Input)
Default: N = size (X,1).

XGUESS — Vector of length N containing the initial guess of the minimum. (Input)
Default: XGUESS = 0.0.

GRADTL — Convergence criterion. (Input)
The calculation ends when the sum of squares of the components of G is less than GRADTL.
Default: GRADTL = 1.e-4.

MAXFN — Maximum number of function evaluations. (Input)
Default: MAXFN = 100.

G — Vector of length N containing the components of the gradient at the final parameter estimates. (Output)

FVALUE — Scalar containing the value of the function at the computed solution. (Output)

FORTRAN 90 Interface

Generic: CALL UMCGG (FCN, GRAD, DFPRED, X [,…])

Specific: The specific interface names are S_UMCGG and D_UMCGG.

FORTRAN 77 Interface

Single: CALL UMCGG (FCN, GRAD, N, XGUESS, GRADTL, MAXFN, DFPRED, X, G, FVALUE)

Double: The double precision name is DUMCGG.

Example

The function

$$f\left(x\right)=100\left(x_2-x_1^2\right)^2+\left(1-x_1\right)^2$$

is minimized and the solution is printed.

```
      USE UMCGG_INT
      USE UMACH_INT
!                                 Declaration of variables
      INTEGER    N
      PARAMETER  (N=2)
!
      INTEGER    I, NOUT
      REAL       DFPRED, FVALUE, G(N), GRADTL, X(N), &
                XGUESS(N)
      EXTERNAL   ROSBRK, ROSGRD
!
      DATA XGUESS/-1.2E0, 1.0E0/
!
      DFPRED = 0.2
      GRADTL = 1.0E-7
!                                 Minimize the Rosenbrock function
      CALL UMCGG (ROSBRK, ROSGRD, DFPRED, X, XGUESS=XGUESS, &
                 GRADTL=GRADTL, G=G, FVALUE=FVALUE)
!                                 Print the results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) (X(I),I=1,N), FVALUE, (G(I),I=1,N)
99999 FORMAT (’  The solution is ’, 2F8.3, //, ’  The function ’, &
            ’evaluated at the solution is ’, F8.3, //, ’  The ’, &
            ’gradient is ’, 2F8.3, /)
!
      END
!
      SUBROUTINE ROSBRK (N, X, F)
      INTEGER    N
      REAL       X(N), F
!
      F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
      RETURN
      END
!
      SUBROUTINE ROSGRD (N, X, G)
      INTEGER    N
      REAL       X(N), G(N)
!
      G(1) = -4.0E2*(X(2)-X(1)*X(1))*X(1) - 2.0E0*(1.0E0-X(1))
      G(2) = 2.0E2*(X(2)-X(1)*X(1))
!
      RETURN
      END
```

Output

```
The solution is    1.000   1.000

The function evaluated at the solution is    0.000

The gradient is    0.000   0.000
```

Comments

1. Workspace may be explicitly provided, if desired, by use of U2CGG/DU2CGG. The reference is:

   ```
   CALL U2CGG (FCN, GRAD, N, XGUESS, GRADTL, MAXFN, DFPRED, X, G,
   FVALUE, S, RSS, RSG, GINIT, XOPT, GOPT)
   ```

   The additional arguments are as follows:

   S — Vector of length N used for the search direction in each iteration.

   RSS — Vector of length N containing conjugacy information.

   RSG — Vector of length N containing conjugacy information.

   GINIT — Vector of length N containing the gradient values at the start on an iteration.

   XOPT — Vector of length N containing the parameter values which yield the least calculated value for FVALUE.

   GOPT — Vector of length N containing the gradient values which yield the least calculated value for FVALUE.

2. Informational errors

Type	Code	
4	1	The line search of an integration was abandoned. This error may be caused by an error in gradient.
4	2	The calculation cannot continue because the search is uphill.
4	3	The iteration was terminated because MAXFN was exceeded.
3	4	The calculation was terminated because two consecutive iterations failed to reduce the function.

3. The routine includes no thorough checks on the part of the user program that calculates the derivatives of the objective function. Therefore, because derivative calculation is a frequent source of error, the user should verify independently the correctness of the derivatives that are given to the routine.

4. Because of the close relation between the conjugate-gradient method and the method of steepest descent, it is very helpful to choose the scale of the variables in a way that balances the magnitudes of the components of a typical gradient vector. It can be particularly inefficient if a few components of the gradient are much larger than the rest.

5. If the value of the parameter GRADTL in the argument list of the routine is set to zero, then the subroutine will continue its calculation until it stops reducing the objective function. In this case, the usual behavior is that changes in the objective function become dominated by computer rounding errors before precision is lost in the gradient vector. Therefore, because the point of view has been taken that the user requires the least possible value of the function, a value of the objective function that is small due to computer rounding errors can prevent further progress. Hence, the precision in the final values of the variables may be only about half the number of significant digits in the computer arithmetic, but the least value of FVALUE is usually found to be quite accurate.

Description

The routine UMCGG uses a conjugate gradient method to find the minimum of a function $f(x)$ of n variables. Function values and first derivatives are required.

The routine is based on the version of the conjugate gradient algorithm described in Powell (1977). The main advantage of the conjugate gradient technique is that it provides a fast rate of convergence without the storage of any matrices. Therefore, it is particularly suitable for unconstrained minimization calculations where the number of variables is so large that matrices of dimension n cannot be stored in the main memory of the computer. For smaller problems, however, a subroutine such as IMSL routine UMING (page 1202), is usually more efficient because each iteration makes use of additional information from previous iterations.

UMPOL

Minimizes a function of N variables using a direct search polytope algorithm.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function to be minimized. The usage is CALL FCN (N, X, F), where

N – Length of X. (Input)

X – Vector of length N at which point the function is evaluated. (Input)
X should not be changed by FCN.

F – The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

X — Real vector of length N containing the best estimate of the minimum found. (Output)

Optional Arguments

N — Dimension of the problem. (Input)
Default: N = size (X,1).

XGUESS — Real vector of length N which contains an initial guess to the minimum. (Input)
Default: XGUESS = 0.0.

S — On input, real scalar containing the length of each side of the initial simplex. (Input/Output)
If no reasonable information about S is known, S could be set to a number less than or equal to zero and UMPOL will generate the starting simplex from the initial guess with a random number generator. On output, the average distance from the vertices to the centroid that is taken to be the solution; see Comment 4.
Default: S = 0.0.

FTOL — First convergence criterion. (Input)
The algorithm stops when a relative error in the function values is less than FTOL, i.e. when (F(worst) − F(best)) < FTOL * (1 + ABS(F(best))) where F(worst) and F(best) are the function values of the current worst and best points, respectively. Second convergence criterion. The algorithm stops when the standard deviation of the function values at the N + 1 current points is less than FTOL. If the subroutine terminates prematurely, try again with a smaller value for FTOL.
Default: FTOL = 1.e-7.

MAXFCN — On input, maximum allowed number of function evaluations. (Input/ Output)
On output, actual number of function evaluations needed.
Default: MAXFCN = 200.

FVALUE — Function value at the computed solution. (Output)

FORTRAN 90 Interface

Generic: CALL UMPOL (FCN, X [,…])

Specific: The specific interface names are S_UMPOL and D_UMPOL.

FORTRAN 77 Interface

Single: CALL UMPOL (FCN, N, XGUESS, S, FTOL, MAXFCN, X, FVALUE)

Double: The double precision name is DUMPOL.

Example

The function

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + \left(1 - x_1\right)^2$$

is minimized and the solution is printed.

```
      USE UMPOL_INT
      USE UMACH_INT
!                                  Variable declarations
```

```
      INTEGER    N
      PARAMETER  (N=2)
!
      INTEGER    K, NOUT
      REAL       FTOL, FVALUE, S, X(N), XGUESS(N)
      EXTERNAL   FCN
!
!                                  Initializations
!                                  XGUESS = ( -1.2, 1.0)
!
      DATA XGUESS/-1.2, 1.0/
!
      FTOL   = 1.0E-10
      S      = 1.0
!
      CALL UMPOL (FCN, X, XGUESS=XGUESS, S=S, FTOL=FTOL,&
                  FVALUE=FVALUE)
!
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) (X(K),K=1,N), FVALUE
99999 FORMAT ('  The best estimate for the minimum value of the', /, &
            '  function is X = (', 2(2X,F4.2), ')', /, '  with ', &
            'function value FVALUE = ', E12.6)
!
      END
!                                  External function to be minimized
      SUBROUTINE FCN (N, X, F)
      INTEGER    N
      REAL       X(N), F
!
      F = 100.0*(X(1)*X(1)-X(2))**2 + (1.0-X(1))**2
      RETURN
      END
```

Output

```
The best estimate for the minimum value of the
function is X = (  1.00  1.00)
with function value FVALUE = 0.502496E-10
```

Comments

1. Workspace may be explicitly provided, if desired, by use of U2POL/DU2POL. The reference is:

   ```
   CALL U2POL (FCN, N, XGUESS, S, FTOL, MAXFCN, X,
   FVALUE, WK)
   ```

   The additional argument is:

   WK — Real work vector of length N**2 + 5 * N + 1.

2. Informational error

   Type Code

4	1	Maximum number of function evaluations exceeded.

3. Since UMPOL uses only function value information at each step to determine a new approximate minimum, it could be quite ineficient on smooth problems compared to other methods such as those implemented in routine UMINF that takes into account derivative information at each iteration. Hence, routine UMPOL should only be used as a last resort. Briefly, a set of N + 1 points in an N-dimensional space is called a simplex. The minimization process iterates by replacing the point with the largest function value by a new point with a smaller function value. The iteration continues until all the points cluster sufficiently close to a minimum.

4. The value returned in *S* is useful for assessing the flatness of the function near the computed minimum. The larger its value for a given value of FTOL, the flatter the function tends to be in the neighborhood of the returned point.

Description

The routine UMPOL uses the polytope algorithm to find a minimum point of a function $f(x)$ of n variables. The polytope method is based on function comparison; no smoothness is assumed. It starts with $n + 1$ points $x_1, x_2, \ldots, x_{n+1}$. At each iteration, a new point is generated to replace the worst point x_j, which has the largest function value among these $n + 1$ points. The new point is constructed by the following formula:

$$x_k = c + \alpha(c - x_j)$$

where

$$c = \frac{1}{n}\sum_{i \neq j} x_i$$

and α ($\alpha > 0$) is the *reflection coefficient.*

When x_k is a best point, that is $f(x_k) \le f(x_i)$ for $i = 1, \ldots, n + 1$, an expansion point is computed $x_e = c + \beta(x_k - c)$ where $\beta(\beta > 1)$ is called the *expansion coefficient*. If the new point is a worst point, then the polytope would be contracted to get a better new point. If the contraction step is unsuccessful, the polytope is shrunk by moving the vertices halfway toward current best point. This procedure is repeated until one of the following stopping criteria is satisfied:

Criterion 1:

$$f_{best} - f_{worst} \le \varepsilon_f(1. + |f_{best}|)$$

Criterion 2:

$$\sum_{i=1}^{n+1}\left(f_i - \frac{\sum_{j=1}^{n+1} f_j}{n+1}\right)^2 \le \varepsilon_f$$

where $f_i = f(x_i)$, $f_j = f(x_j)$, and ε_f is a given tolerance. For a complete description, see Nelder and Mead (1965) or Gill et al. (1981).

UNLSF

Solves a nonlinear least-squares problem using a modified Levenberg-Marquardt algorithm and a finite-difference Jacobian.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function that defines the least-squares problem. The usage is CALL FCN (M, N, X, F), where

M – Length of F. (Input)

N – Length of X. (Input)

X – Vector of length N at which point the function is evaluated. (Input)
X should not be changed by FCN.

F – Vector of length M containing the function values at X. (Output)

FCN must be declared EXTERNAL in the calling program.

M — Number of functions. (Input)

X — Vector of length N containing the approximate solution. (Output)

Optional Arguments

N — Number of variables. N must be less than or equal to M. (Input)
Default: N = size (X,1).

XGUESS — Vector of length N containing the initial guess. (Input)
Default: NDEG = size (COEFF,1) – 1.

XSCALE — Vector of length N containing the diagonal scaling matrix for the variables. (Input)
XSCALE is used mainly in scaling the gradient and the distance between two points. By default, the values for XSCALE are set internally. See IPARAM(6) in Comment 4.
Default: XSCALE = 1.0.

FSCALE — Vector of length M containing the diagonal scaling matrix for the functions. (Input)
FSCALE is used mainly in scaling the gradient. In the absence of other information, set all entries to 1.0.
Default: FSCALE = 1.0.

IPARAM — Parameter vector of length 6. (Input/Output)
Set `IPARAM`(1) to zero for default values of `IPARAM` and `RPARAM`. See Comment 4.
Default: `IPARAM` = 0.

RPARAM — Parameter vector of length 7. (Input/Output)
See Comment 4.

FVEC — Vector of length `M` containing the residuals at the approximate solution. (Output)

FJAC — `M` by `N` matrix containing a finite difference approximate Jacobian at the approximate solution. (Output)

LDFJAC — Leading dimension of `FJAC` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDFJAC` = size (`FJAC`,1).

FORTRAN 90 Interface

Generic: `CALL UNLSF (FCN, M, X [,…])`

Specific: The specific interface names are `S_UNLSF` and `D_UNLSF`.

FORTRAN 77 Interface

Single:
```
CALL UNLSF (FCN, M, N, XGUESS, XSCALE, FSCALE, IPARAM,
RPARAM, X, FVEC, FJAC, LDFJAC)
```

Double: The double precision name is `DUNLSF`.

Example

The nonlinear least squares problem

$$\min_{x \in \mathbf{R}^2} \frac{1}{2} \sum_{i=1}^{2} f_i(x)^2$$

where

$$f_1(x) = 10\left(x_2 - x_1^2\right) \text{ and } f_2(x) = \left(1 - x_1\right)$$

is solved. `RPARAM`(4) is changed to a non-default value.

```
      USE UNLSF_INT
      USE UMACH_INT
      USE U4LSF_INT
!                                  Declaration of variables
      INTEGER    LDFJAC, M, N
      PARAMETER  (LDFJAC=2, M=2, N=2)
!
      INTEGER    IPARAM(6), NOUT
      REAL       FVEC(M), RPARAM(7),X(N), XGUESS(N)
      EXTERNAL   ROSBCK
!                                 Compute the least squares for the
!                                 Rosenbrock function.
      DATA XGUESS/-1.2E0, 1.0E0/
!
!                                 Relax the first stopping criterion by
!                                 calling U4LSF and scaling the
!                                 absolute function tolerance by 10.
      CALL U4LSF (IPARAM, RPARAM)
      RPARAM(4) = 10.0E0*RPARAM(4)
!
      CALL UNLSF (ROSBCK, M, X,XGUESS=XGUESS, IPARAM=IPARAM, &
                 RPARAM=RPARAM, FVEC=FVEC)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) X, FVEC, IPARAM(3), IPARAM(4)
!
99999 FORMAT ('  The solution is ', 2F9.4, //, '  The function ', &
            'evaluated at the solution is ', /, 18X, 2F9.4, //, &
            '  The number of iterations is ', 10X, I3, /, '  The ', &
            'number of function evaluations is ', I3, /)
      END
!
      SUBROUTINE ROSBCK (M, N, X, F)
      INTEGER    M, N
      REAL       X(N), F(M)
!
      F(1) = 10.0E0*(X(2)-X(1)*X(1))
      F(2) = 1.0E0 - X(1)
      RETURN
      END
```

Output

```
The solution is    1.0000   1.0000

The function evaluated at the solution is
0.0000   0.0000

The number of iterations is            24
The number of function evaluations is  33
```

Comments

1. Workspace may be explicitly provided, if desired, by use of U2LSF/DU2LSF. The reference is:

   ```
   CALL U2LSF (FCN, M, N, XGUESS, XSCALE, FSCALE, IPARAM, RPARAM,
   X, FVEC, FJAC, LDFJAC, WK, IWK)
   ```

   The additional arguments are as follows:

   WK — Real work vector of length 9 * N + 3 * M – 1. WK contains the following information on output: The second N locations contain the last step taken. The third N locations contain the last Gauss-Newton step. The fourth N locations contain an estimate of the gradient at the solution.

IWK — Integer work vector of length N containing the permutations used in the QR factorization of the Jacobian at the solution.

2. Informational errors

Type	Code	
3	1	Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.
3	2	The iterates appear to be converging to a noncritical point.
4	3	Maximum number of iterations exceeded.
4	4	Maximum number of function evaluations exceeded.
3	6	Five consecutive steps have been taken with the maximum step length.
2	7	Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or STEPTL is too big.

3. The first stopping criterion for UNLSF occurs when the norm of the function is less than the absolute function tolerance (RPARAM(4)). The second stopping criterion occurs when the norm of the scaled gradient is less than the given gradient tolerance (RPARAM(1)). The third stopping criterion for UNLSF occurs when the scaled distance between the last two steps is less than the step tolerance (RPARAM(2)).

4. If the default parameters are desired for UNLSF, then set IPARAM(1) to zero and call the routine UNLSF. Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM, then the following steps should be taken before calling UNLSF:

CALL U4LSF (IPARAM, RPARAM)
Set nondefault values for desired IPARAM, RPARAM elements.

Note that the call to U4LSF will set IPARAM and RPARAM to their default values so only nondefault values need to be set above.

The following is a list of the parameters and the default values:

IPARAM — Integer vector of length 6.

IPARAM(1) = Initialization flag.

IPARAM(2) = Number of good digits in the function.
Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.
Default: 100.

IPARAM(4) = Maximum number of function evaluations.
Default: 400.

IPARAM(5) = Maximum number of Jacobian evaluations.
Default: Not used in UNLSF.

IPARAM(6) = Internal variable scaling flag.
If IPARAM(6) = 1, then the values for XSCALE are set internally.
Default: 1.

RPARAM — Real vector of length 7.
RPARAM(1) = Scaled gradient tolerance.
The *i*-th component of the scaled gradient at x is calculated as

$$\frac{|g_i| * \max(|x_i|, 1/s_i)}{\|F(x)\|_2^2}$$

where

$$g_i = \left(J(x)^T F(x)\right)_i * (f_s)_i^2$$

J(*x*) is the Jacobian, *s* = XSCALE, and f_s = FSCALE.
Default:

$$\sqrt{\varepsilon}, \sqrt[3]{\varepsilon}$$

in double where ε is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)
The *i*-th component of the scaled step between two points *x* and *y* is computed as

$$\frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)}$$

where *s* = XSCALE.
Default: $\varepsilon^{2/3}$ where ε is the machine precision.

RPARAM(3) = Relative function tolerance.
Default: max(10^{-10}, $\varepsilon^{2/3}$), max (10^{-20}, $\varepsilon^{2/3}$) in double where ε is the machine precision.

RPARAM(4) = Absolute function tolerance.
Default: max (10^{-20}, ε^2), max(10^{-40}, ε^2) in double where ε is the machine precision.

RPARAM(5) = False convergence tolerance.
Default: 100ε where ε is the machine precision.

RPARAM(6) = Maximum allowable step size.
Default: 1000 max(ε_1, ε_2) where

$$\varepsilon_1 = \sqrt{\sum_{i=1}^{n} \left(s_i t_i\right)^2}$$

$\varepsilon_2 = \| s \|_2$, s = XSCALE, and t = XGUESS.

RPARAM(7) = Size of initial trust region radius.
Default: based on the initial scaled Cauchy step.

If double precision is desired, then DU4LSF is called and RPARAM is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to “Error Handling” in the Introduction.

Description

The routine UNLSF is based on the MINPACK routine LMDIF by Moré et al. (1980). It uses a modified Levenberg-Marquardt method to solve nonlinear least squares problems. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} \frac{1}{2} F(x)^T F(x) = \frac{1}{2} \sum_{i=1}^{m} f_i(x)^2$$

where $m \ge n$, $F : \mathbf{R}^n \rightarrow \mathbf{R}^m$, and $f_i(x)$ is the i-th component function of $F(x)$. From a current point, the algorithm uses the trust region approach:

$$\min_{x_n \in \mathbf{R}^n} \left\| F(x_c) + J(x_c)(x_n - x_c) \right\|_2$$

$$\text{subject to } \|x_n - x_c\|_2 \le \delta_c$$

to get a new point x_n, which is computed as

$$x_n = x_c - \left(J(x_c)^T J(x_c) + \mu_c I \right)^{-1} J(x_c)^T F(x_c)$$

where $\mu_c = 0$ if $\delta_c \ge \|(J(x_c)^T J(x_c))^{-1} J(x_c)^T F(x_c)\|_2$ and $\mu_c > 0$ otherwise. $F(x_c)$ and $J(x_c)$ are the function values and the Jacobian evaluated at the current point x_c. This procedure is repeated until the stopping criteria are satisfied. For more details, see Levenberg (1944), Marquardt (1963), or Dennis and Schnabel (1983, Chapter 10).

Since a finite-difference method is used to estimate the Jacobian for some single precision calculations, an inaccurate estimate of the Jacobian may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended. Also, whenever the exact Jacobian can be easily provided, routine UNLSJ (page 1237) should be used instead.

UNLSJ

Solves a nonlinear least squares problem using a modified Levenberg-Marquardt algorithm and a user-supplied Jacobian.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function which defines the least-squares problem. The usage is CALL FCN (M, N, X, F), where

M – Length of F. (Input)
N – Length of X. (Input)
X – Vector of length N at which point the function is evaluated. (Input)
X should not be changed by FCN.
F – Vector of length M containing the function values at X. (Output)

FCN must be declared EXTERNAL in the calling program.

JAC — User-supplied SUBROUTINE to evaluate the Jacobian at a point X. The usage is CALL JAC (M, N, X, FJAC, LDFJAC), where

M – Length of F. (Input)
N – Length of X. (Input)
X – Vector of length N at which point the Jacobian is evaluated. (Input)
X should not be changed by JAC.
FJAC – The computed M by N Jacobian at the point X. (Output)
LDFJAC – Leading dimension of FJAC. (Input)

JAC must be declared EXTERNAL in the calling program.

M — Number of functions. (Input)

X — Vector of length N containing the approximate solution. (Output)

Optional Arguments

N — Number of variables. N must be less than or equal to M. (Input)
Default: N = size (X,1).

XGUESS — Vector of length N containing the initial guess. (Input)
Default: XGUESS = 0.0.

XSCALE — Vector of length N containing the diagonal scaling matrix for the variables. (Input)
XSCALE is used mainly in scaling the gradient and the distance between two points. By default, the values for XSCALE are set internally. See IPARAM(6) in Comment 4.
Default: XSCALE = 1.0.

FSCALE — Vector of length `M` containing the diagonal scaling matrix for the functions. (Input)
`FSCALE` is used mainly in scaling the gradient. In the absence of other information, set all entries to 1.0.
Default: `FSCALE` = 1.0.

IPARAM — Parameter vector of length 6. (Input/Output)
Set `IPARAM`(1) to zero for default values of `IPARAM` and `RPARAM`. See Comment 4.
Default: `IPARAM` = 0.

RPARAM — Parameter vector of length 7. (Input/Output)
See Comment 4.

FVEC — Vector of length `M` containing the residuals at the approximate solution. (Output)

FJAC — `M` by `N` matrix containing a finite-difference approximate Jacobian at the approximate solution. (Output)

LDFJAC — Leading dimension of `FJAC` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDFJAC` = size (`FJAC`,1).

FORTRAN 90 Interface

Generic: `CALL UNLSJ (FCN, JAC, M, X [,…])`

Specific: The specific interface names are `S_UNLSJ` and `D_UNLSJ`.

FORTRAN 77 Interface

Single:
```
CALL UNLSJ (FCN, JAC, M, N, XGUESS, XSCALE, FSCALE, IPARAM,
RPARAM, X, FVEC, FJAC, LDFJAC)
```

Double: The double precision name is `DUNLSJ`.

Example

The nonlinear least-squares problem

$$\min_{x\in\mathbf{R}^2}\frac{1}{2}\sum_{i=1}^{2}f_i\left(x\right)^2$$

where

$$f_1\left(x\right)=10\left(x_2-x_1^2\right)\text{ and }f_2\left(x\right)=\left(1-x_1\right)$$

is solved; default values for parameters are used.

```
      USE UNLSJ_INT
      USE UMACH_INT
!                                 Declaration of variables
      INTEGER    LDFJAC, M, N
      PARAMETER  (LDFJAC=2, M=2, N=2)
!
      INTEGER    IPARAM(6), NOUT
      REAL       FVEC(M), X(N), XGUESS(N)
      EXTERNAL   ROSBCK, ROSJAC
!                                 Compute the least squares for the
!                                 Rosenbrock function.
      DATA XGUESS/-1.2E0, 1.0E0/
      IPARAM(1) = 0
!
      CALL UNLSJ (ROSBCK, ROSJAC, M, X, XGUESS=XGUESS, &
                 IPARAM=IPARAM, FVEC=FVEC)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) X, FVEC, IPARAM(3), IPARAM(4), IPARAM(5)
!
99999 FORMAT ('  The solution is ', 2F9.4, //, '  The function ', &
            'evaluated at the solution is ', /, 18X, 2F9.4, //, &
            '  The number of iterations is ', 10X, I3, /, '  The ', &
            'number of function evaluations is ', I3, /, '  The ', &
            'number of Jacobian evaluations is ', I3, /)
      END
!
      SUBROUTINE ROSBCK (M, N, X, F)
      INTEGER    M, N
      REAL       X(N), F(M)
!
      F(1) = 10.0E0*(X(2)-X(1)*X(1))
      F(2) = 1.0E0 - X(1)
      RETURN
      END
!
      SUBROUTINE ROSJAC (M, N, X, FJAC, LDFJAC)
      INTEGER    M, N, LDFJAC
      REAL       X(N), FJAC(LDFJAC,N)
!
      FJAC(1,1) = -20.0E0*X(1)
      FJAC(2,1) = -1.0E0
      FJAC(1,2) = 10.0E0
      FJAC(2,2) = 0.0E0
      RETURN
      END
```

Output

```
The solution is    1.0000   1.0000

The function evaluated at the solution is
0.0000   0.0000

The number of iterations is            23
The number of function evaluations is  32
The number of Jacobian evaluations is  24
```

Comments

1. Workspace may be explicitly provided, if desired, by use of U2LSJ/DU2LSJ. The reference is:

   ```
   CALL U2LSJ (FCN, JAC, M, N, XGUESS, XSCALE, FSCALE, IPARAM,
   RPARAM, X, FVEC, FJAC, LDFJAC, WK, IWK)
   ```

   The additional arguments are as follows:

   WK — Work vector of length 9 * N + 3 * M – 1. WK contains the following information on output: The second N locations contain the last step taken. The third N locations contain the last Gauss-Newton step. The fourth N locations contain an estimate of the gradient at the solution.

   IWK — Work vector of length N containing the permutations used in the QR factorization of the Jacobian at the solution.

2. Informational errors

Type	Code	
3	1	Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.
3	2	The iterates appear to be converging to a noncritical point.
4	3	Maximum number of iterations exceeded.
4	4	Maximum number of function evaluations exceeded.
4	5	Maximum number of Jacobian evaluations exceeded.
3	6	Five consecutive steps have been taken with the maximum step length.
2	7	Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or STEPTL is too big.

3. The first stopping criterion for UNLSJ occurs when the norm of the function is less than the absolute function tolerance (RPARAM(4)). The second stopping criterion occurs when the norm of the scaled gradient is less than the given gradient tolerance (RPARAM(1)). The third stopping criterion for UNLSJ occurs when the scaled distance between the last two steps is less than the step tolerance (RPARAM(2)).

4. If the default parameters are desired for UNLSJ, then set IPARAM(1) to zero and call the routine UNLSJ. Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM, then the following steps should be taken before calling UNLSJ:

   CALL U4LSF (IPARAM, RPARAM)
   Set nondefault values for desired IPARAM, RPARAM elements.

Note that the call to U4LSF will set IPARAM and RPARAM to their default values, so only nondefault values need to be set above.

The following is a list of the parameters and the default values:

IPARAM — Integer vector of length 6.
IPARAM(1) = Initialization flag.

IPARAM(2) = Number of good digits in the function.
Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.
Default: 100.

IPARAM(4) = Maximum number of function evaluations.
Default: 400.

IPARAM(5) = Maximum number of Jacobian evaluations.
Default: 100.

IPARAM(6) = Internal variable scaling flag.
If IPARAM(6) = 1, then the values for XSCALE are set internally.
Default: 1.

RPARAM — Real vector of length 7.

RPARAM(1) = Scaled gradient tolerance.
The *i*-th component of the scaled gradient at x is calculated as

$$\frac{|g_i| * \max\left(|x_i|, 1/s_i\right)}{\left\|F(x)\right\|_2^2}$$

where

$$g_i = \left(J(x)^T F(x)\right)_i * \left(f_s\right)_i^2$$

$J(x)$ is the Jacobian, s = XSCALE, and f_s = FSCALE.
Default:

$$\sqrt{\varepsilon}, \sqrt[3]{\varepsilon}$$

in double where ε is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)
The *i*-th component of the scaled step between two points *x* and *y* is computed as

$$\frac{|x_i - y_i|}{\max\left(|x_i|, 1/s_i\right)}$$

where s = `XSCALE`.
Default: $\varepsilon^{2/3}$ where ε is the machine precision.

`RPARAM`(3) = Relative function tolerance.
Default: max(10^{-10}, $\varepsilon^{2/3}$), max (10^{-20}, $\varepsilon^{2/3}$) in double where ε is the machine precision.

`RPARAM`(4) = Absolute function tolerance.
Default: max (10^{-20}, ε^2), max(10^{-40}, ε^2) in double where ε is the machine precision.

`RPARAM`(5) = False convergence tolerance.
Default: 100ε where ε is the machine precision.

`RPARAM`(6) = Maximum allowable step size.
Default: 1000 max(ε_1, ε_2) where

$$\varepsilon_1 \sqrt{\sum\nolimits_{i=1}^{n} \left(s_i t_i\right)^2}$$

$\varepsilon_2 = \| s \|_2$, s = `XSCALE`, and t = `XGUESS`.

`RPARAM`(7) = Size of initial trust region radius.
Default: based on the initial scaled Cauchy step.

If double precision is desired, then `DU4LSF` is called and `RPARAM` is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to "Error Handling" in the Introduction.

Description

The routine `UNLSJ` is based on the MINPACK routine `LMDER` by Moré et al. (1980). It uses a modified Levenberg-Marquardt method to solve nonlinear least squares problems. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} \frac{1}{2} F(x)^T F(x) = \frac{1}{2} \sum_{i=1}^{m} f_i(x)^2$$

where $m \geq n$, $F : \mathbf{R}^n \rightarrow \mathbf{R}^m$, and $f_i(x)$ is the i-th component function of $F(x)$. From a current point, the algorithm uses the trust region approach:

$$\min_{x_n \in \mathbf{R}^n} \left\| F\left(x_c\right) + J\left(x_c\right)\left(x_n - x_c\right) \right\|_2$$

$$\text{subject to } \|x_n - x_c\|_2 \le \delta_c$$

to get a new point x_n, which is computed as

$$x_n = x_c - \left(J\left(x_c\right)^T J\left(x_c\right) + \mu_c I \right)^{-1} J\left(x_c\right)^T F\left(x_c\right)$$

where $\mu_c = 0$ if $\delta_c \ge \|(J(x_c)^T J(x_c))^{-1} J(x_c)^T F(x_c)\|_2$ and $\mu_c > 0$ otherwise. $F(x_c)$ and $J(x_c)$ are the function values and the Jacobian evaluated at the current point x_c. This procedure is repeated until the stopping criteria are satisfied. For more details, see Levenberg (1944), Marquardt(1963), or Dennis and Schnabel (1983, Chapter 10).

BCONF

Minimizes a function of N variables subject to bounds on the variables using a quasi-Newton method and a finite-difference gradient.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function to be minimized. The usage is CALL FCN (N, X, F), where

N – Length of X. (Input)

X – Vector of length N at which point the function is evaluated. (Input)
X should not be changed by FCN.

F – The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

IBTYPE — Scalar indicating the types of bounds on variables. (Input)

IBTYPE	**Action**
0	User will supply all the bounds.
1	All variables are nonnegative.
2	All variables are nonpositive.
3	User supplies only the bounds on 1st variable, all other variables will have the same bounds.

XLB — Vector of length N containing the lower bounds on variables. (Input, if IBTYPE = 0; output, if IBTYPE = 1 or 2; input/output, if IBTYPE = 3)

XUB — Vector of length N containing the upper bounds on variables. (Input, if IBTYPE = 0; output, if IBTYPE = 1 or 2; input/output, if IBTYPE = 3)

X — Vector of length N containing the computed solution. (Output)

Optional Arguments

N — Dimension of the problem. (Input)
Default: N = size (X,1).

XGUESS — Vector of length N containing an initial guess of the computed solution. (Input)
Default: XGUESS = 0.0.

XSCALE — Vector of length N containing the diagonal scaling matrix for the variables. (Input)
XSCALE is used mainly in scaling the gradient and the distance between two points. In the absence of other information, set all entries to 1.0.
Default: XSCALE = 1.0.

FSCALE — Scalar containing the function scaling. (Input)
FSCALE is used mainly in scaling the gradient. In the absence of other information, set FSCALE to 1.0.
Default: FSCALE = 1.0.

IPARAM — Parameter vector of length 7. (Input/Output)
Set IPARAM(1) to zero for default values of IPARAM and RPARAM. See Comment 4.
Default: IPARAM = 0.

RPARAM — Parameter vector of length 7. (Input/Output)
See Comment 4.

FVALUE — Scalar containing the value of the function at the computed solution. (Output)

FORTRAN 90 Interface

Generic: `CALL BCONF (FCN, IBTYPE, XLB, XUB, X [,…])`

Specific: The specific interface names are S_BCONF and D_BCONF.

FORTRAN 77 Interface

Single:
```
CALL BCONF (FCN, N, XGUESS, IBTYPE, XLB, XUB, XSCALE,
 FSCALE, IPARAM, RPARAM, X, FVALUE)
```

Double: The double precision name is DBCONF.

Example

The problem

$$\min f(x) = 100\left(x_2 - x_1^2\right)^2 + \left(1 - x_1\right)^2$$
$$\text{subject to} \quad -2 \le x_1 \le 0.5$$
$$-1 \le x_2 \le 2$$

is solved with an initial guess (−1.2, 1.0) and default values for parameters.

```
      USE BCONF_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=2)
!
      INTEGER    IPARAM(7), ITP, L, NOUT
      REAL       F, FSCALE, RPARAM(7), X(N), XGUESS(N), &
                XLB(N), XSCALE(N), XUB(N)
      EXTERNAL   ROSBRK
!
      DATA XGUESS/-1.2E0, 1.0E0/
      DATA XLB/-2.0E0, -1.0E0/, XUB/0.5E0, 2.0E0/
!                                All the bounds are provided
      ITP = 0
!                                Default parameters are used
      IPARAM(1) = 0
!                                Minimize Rosenbrock function using
!                                initial guesses of -1.2 and 1.0
      CALL BCONF (ROSBRK, ITP, XLB, XUB, X, XGUESS=XGUESS,  &
                 IPARAM=IPARAM, FVALUE=F)
!                                Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) X, F, (IPARAM(L),L=3,5)
!
99999 FORMAT ('  The solution is ', 6X, 2F8.3, //, '  The function ', &
            'value is ', F8.3, //, '  The number of iterations is ', &
            10X, I3, /, '  The number of function evaluations is ', &
            I3, /, '  The number of gradient evaluations is ', I3)
!
      END
!
      SUBROUTINE ROSBRK (N, X, F)
      INTEGER    N
      REAL       X(N), F
!
      F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
!
      RETURN
      END
```

Output

```
The solution is          0.500   0.250

The function value is    0.250
The number of iterations is           24
The number of function evaluations is  34
The number of gradient evaluations is  26
```

Comments

1. Workspace may be explicitly provided, if desired, by use of B2ONF/DB2ONF. The reference is:

   ```
   CALL B2ONF (FCN, N, XGUESS, IBTYPE, XLB, XUB,
   XSCALE, FSCALE, IPARAM, RPARAM, X, FVALUE, WK, IWK)
   ```

   The additional arguments are as follows:

   WK — Real work vector of length N * (2 * N + 8). WK contains the following information on output: The second N locations contain the last step taken. The third N locations contain the last Newton step. The fourth N locations contain an estimate of the gradient at the solution. The final N^2 locations contain a BFGS approximation to the Hessian at the solution.

   IWK — Work vector of length N stored in column order. Only the lower triangular portion of the matrix is stored in WK. The values returned in the upper triangle should be ignored.

2. Informational errors

Type	Code	
3	1	Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.
4	2	The iterates appear to be converging to a noncritical point.
4	3	Maximum number of iterations exceeded.
4	4	Maximum number of function evaluations exceeded.
4	5	Maximum number of gradient evaluations exceeded.
4	6	Five consecutive steps have been taken with the maximum step length.
2	7	Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or STEPTL is too big.
3	8	The last global step failed to locate a lower point than the current X value.

3. The first stopping criterion for BCONF occurs when the norm of the gradient is less than the given gradient tolerance (RPARAM(1)). The second stopping criterion for BCONF occurs when the scaled distance between the last two steps is less than the step tolerance (RPARAM(2)).

4. If the default parameters are desired for BCONF, then set IPARAM(1) to zero and call the routine BCONF. Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM, then the following steps should be taken before calling BCONF:

CALL U4INF (IPARAM, RPARAM)

Set nondefault values for desired IPARAM, RPARAM elements.

Note that the call to U4INF will set IPARAM and RPARAM to their default values so only nondefault values need to be set above.

The following is a list of the parameters and the default values:

IPARAM — Integer vector of length 7.
IPARAM(1) = Initialization flag.

IPARAM(2) = Number of good digits in the function.
Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.
Default: 100.

IPARAM(4) = Maximum number of function evaluations.
Default: 400.

IPARAM(5) = Maximum number of gradient evaluations.
Default: 400.

IPARAM(6) = Hessian initialization parameter.
If IPARAM(6) = 0, the Hessian is initialized to the identity matrix; otherwise, it is initialized to a diagonal matrix containing

$$\max\left(\left|f\left(t\right)\right|, f_s\right) * s_i^2$$

on the diagonal where t = XGUESS, f_s = FSCALE, and s = XSCALE.
Default: 0.

IPARAM(7) = Maximum number of Hessian evaluations.
Default: Not used in BCONF.

RPARAM — Real vector of length 7.
RPARAM(1) = Scaled gradient tolerance.
The *i*-th component of the scaled gradient at x is calculated as

$$\frac{\left|g_i\right| * \max\left(\left|x_i\right|, 1/s_i\right)}{\max\left(\left|f\left(x\right)\right|, f_s\right)}$$

where $g = \nabla f(x)$, s = XSCALE, and f_s = FSCALE.
Default:

$$\sqrt{\varepsilon}, \sqrt[3]{\varepsilon}$$

in double where ε is the machine precision.

`RPARAM`(2) = Scaled step tolerance. (`STEPTL`)
The *i*-th component of the scaled step between two points *x* and *y* is computed as

$$\frac{|x_i - y_i|}{\max\left(|x_i|, 1/s_i\right)}$$

where *s* = `XSCALE`.
Default: $\varepsilon^{2/3}$ where ε is the machine precision.

`RPARAM`(3) = Relative function tolerance.
Default: max(10^{-10}, $\varepsilon^{2/3}$), max (10^{-20}, $\varepsilon^{2/3}$) in double where ε is the machine precision.

`RPARAM`(4) = Absolute function tolerance.
Default: Not used in `BCONF`.

`RPARAM`(5) = False convergence $^{\tau o\lambda\varepsilon\rho\alpha\nu\chi\varepsilon}$.
Default: 100ε where ε is the machine precision.

`RPARAM`(6) = Maximum allowable step size.
Default: 1000 max(ε_1, ε_2) where

$$\varepsilon_1 \sqrt{\sum_{i=1}^{n} \left(s_i t_i\right)^2}$$

$\varepsilon_2 = \| s \|_2$, *s* = `XSCALE`, and *t* = `XGUESS`.

`RPARAM`(7) = Size of initial trust region radius.
Default: based on the initial scaled Cauchy step.

If double precision is required, then `DU4INF` is called and `RPARAM` is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to “Error Handling” in the Introduction.

Description

The routine `BCONF` uses a quasi-Newton method and an active set strategy to solve minimization problems subject to simple bounds on the variables. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} f\left(x\right)$$

$$\text{subject to } l \le x \le u$$

From a given starting point x^c, an active set IA, which contains the indices of the variables at their bounds, is built. A variable is called a "free variable" if it is not in the active set. The routine then computes the search direction for the free variables according to the formula

$$d = -B^{-1} g^c$$

where B is a positive definite approximation of the Hessian and g^c is the gradient evaluated at x^c; both are computed with respect to the free variables. The search direction for the variables in IA is set to zero. A line search is used to find a new point x^n ,

$$x^n = x^c + \lambda d, \lambda \in (0, 1]$$

such that

$$f(x^n) \le f(x^c) + \alpha g^T d, \qquad \alpha \in (0, 0.5)$$

Finally, the optimality conditions

$$\|g(x_i)\| \le \varepsilon, \; l_i < x_i < u_i$$

$$g(x_i) < 0, \;\; x_i = u_i$$

$$g(x_i) > 0, \; x_i = l_i$$

are checked, where ε is a gradient tolerance. When optimality is not achieved, B is updated according to the BFGS formula:

$$B \leftarrow B - \frac{Bss^T B}{s^T Bs} + \frac{yy^T}{y^T s}$$

where $s = x^n - x^c$ and $y = g^n - g^c$. Another search direction is then computed to begin the next iteration.

The active set is changed only when a free variable hits its bounds during an iteration or the optimality condition is met for the free variables but not for all variables in IA, the active set. In the latter case, a variable that violates the optimality condition will be dropped out of IA. For more details on the quasi-Newton method and line search, see Dennis and Schnabel (1983). For more detailed information on active set strategy, see Gill and Murray (1976).

Since a finite-difference method is used to estimate the gradient for some single precision calculations, an inaccurate estimate of the gradient may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended. Also, whenever the exact gradient can be easily provided, routine BCONG (page 1249) should be used instead.

BCONG

Minimizes a function of N variables subject to bounds on the variables using a quasi-Newton method and a user-supplied gradient.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function to be minimized. The usage is CALL FCN (N, X, F), where

N – Length of X. (Input)

X – Vector of length N at which point the function is evaluated. (Input)
X should not be changed by FCN.

F – The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

GRAD — User-supplied SUBROUTINE to compute the gradient at the point X. The usage is CALL GRAD (N, X, G), where

N – Length of X and G. (Input)

X – Vector of length N at which point the gradient is evaluated. (Input)
X should not be changed by GRAD.

G – The gradient evaluated at the point X. (Output)

GRAD must be declared EXTERNAL in the calling program.

IBTYPE — Scalar indicating the types of bounds on variables. (Input)

IBTYPE	Action
0	User will supply all the bounds.
1	All variables are nonnegative.
2	All variables are nonpositive.
3	User supplies only the bounds on 1st variable, all other variables will have the same bounds.

XLB — Vector of length N containing the lower bounds on variables. (Input, if IBTYPE = 0; output, if IBTYPE = 1 or 2; input/output, if IBTYPE = 3)

XUB — Vector of length N containing the upper bounds on variables. (Input, if IBTYPE = 0; output, if IBTYPE = 1 or 2; input/output, if IBTYPE = 3)

X — Vector of length N containing the computed solution. (Output)

Optional Arguments

N — Dimension of the problem. (Input)
Default: N = size (X,1).

XGUESS — Vector of length N containing the initial guess of the minimum. (Input)
Default: XGUESS = 0.0.

XSCALE — Vector of length N containing the diagonal scaling matrix for the variables. (Input)
XSCALE is used mainly in scaling the gradient and the distance between two points. In the absence of other information, set all entries to 1.0.
Default: XSCALE = 1.0.

FSCALE — Scalar containing the function scaling. (Input)
FSCALE is used mainly in scaling the gradient. In the absence of other information, set FSCALE to 1.0.
Default: FSCALE = 1.0.

IPARAM — Parameter vector of length 7. (Input/Output)
Set IPARAM(1) to zero for default values of IPARAM and RPARAM. See Comment 4.
Default: IPARAM = 0.

RPARAM — Parameter vector of length 7. (Input/Output)
See Comment 4.

FVALUE — Scalar containing the value of the function at the computed solution. (Output)

FORTRAN 90 Interface

Generic: CALL BCONG (FCN, GRAD, IBTYPE, XLB, XUB, X [,…])

Specific: The specific interface names are S_BCONG and D_BCONG.

FORTRAN 77 Interface

Single: CALL BCONG (FCN, GRAD, N, XGUESS, IBTYPE, XLB, XUB, XSCALE, FSCALE, IPARAM, RPARAM, X, FVALUE)

Double: The double precision name is DBCONG.

Example

The problem

$$\min f\left(x\right) = 100\left(x_2 - x_1^2\right)^2 + \left(1 - x_1\right)^2$$
$$\text{subject to} \quad -2 \le x_1 \le 0.5$$
$$-1 \le x_2 \le 2$$

is solved with an initial guess (–1.2, 1.0), and default values for parameters.

```
      USE BCONG_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=2)
!
      INTEGER    IPARAM(7), ITP, L, NOUT
      REAL       F, X(N), XGUESS(N), XLB(N), XUB(N)
      EXTERNAL   ROSBRK, ROSGRD
!
      DATA XGUESS/-1.2E0, 1.0E0/
      DATA XLB/-2.0E0, -1.0E0/, XUB/0.5E0, 2.0E0/
!                                 All the bounds are provided
      ITP = 0
!                                 Default parameters are used
      IPARAM(1) = 0
!                                 Minimize Rosenbrock function using
!                                 initial guesses of -1.2 and 1.0
      CALL BCONG (ROSBRK, ROSGRD, ITP, XLB, XUB, X, XGUESS=XGUESS, &
                 IPARAM=IPARAM, FVALUE=F)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) X, F, (IPARAM(L),L=3,5)
!
99999 FORMAT ('  The solution is ', 6X, 2F8.3, //, '  The function ', &
            'value is ', F8.3, //, '  The number of iterations is ', &
            10X, I3, /, '  The number of function evaluations is ', &
            I3, /, '  The number of gradient evaluations is ', I3)
!
      END
!
      SUBROUTINE ROSBRK (N, X, F)
      INTEGER    N
      REAL       X(N), F
!
      F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
!
      RETURN
      END
!
      SUBROUTINE ROSGRD (N, X, G)
      INTEGER    N
      REAL       X(N), G(N)
!
      G(1) = -4.0E2*(X(2)-X(1)*X(1))*X(1) - 2.0E0*(1.0E0-X(1))
      G(2) = 2.0E2*(X(2)-X(1)*X(1))
!
      RETURN
      END
```

Output

```
The solution is          0.500   0.250

The function value is    0.250

The number of iterations is            22
The number of function evaluations is  32
The number of gradient evaluations is  23
```

Comments

1. Workspace may be explicitly provided, if desired, by use of B2ONG/DB2ONG. The reference is:

   CALL B2ONG (FCN, GRAD, N, XGUESS, IBTYPE, XLB, XUB, XSCALE, FSCALE, IPARAM, RPARAM, X, FVALUE, WK, IWK)

   The additional arguments are as follows:

   WK — Real work vector of length N * (2 * N + 8). WK contains the following information on output: The second N locations contain the last step taken. The third N locations contain the last Newton step. The fourth N locations contain an estimate of the gradient at the solution. The final N^2 locations contain a BFGS approximation to the Hessian at the solution.

   IWK — Work vector of length N stored in column order. Only the lower triangular portion of the matrix is stored in WK. The values returned in the upper triangle should be ignored.

2. Informational errors

   Type Code

   3 1 Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.

   4 2 The iterates appear to be converging to a noncritical point.

   4 3 Maximum number of iterations exceeded.

   4 4 Maximum number of function evaluations exceeded.

   4 5 Maximum number of gradient evaluations exceeded.

   4 6 Five consecutive steps have been taken with the maximum step length.

   2 7 Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or STEPTL is too big.

3 8 The last global step failed to locate a lower point than the current X value.

3. The first stopping criterion for BCONG occurs when the norm of the gradient is less than the given gradient tolerance (RPARAM(1)). The second stopping criterion for BCONG occurs when the scaled distance between the last two steps is less than the step tolerance (RPARAM(2)).

4. If the default parameters are desired for BCONG, then set IPARAM (1) to zero and call the routine BCONG. Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM, then the following steps should be taken before calling BCONG:

```
CALL U4INF (IPARAM, RPARAM)
```

Set nondefault values for desired IPARAM, RPARAM elements.

Note that the call to U4INF will set IPARAM and RPARAM to their default values so only nondefault values need to be set above.

The following is a list of the parameters and the default values:

IPARAM — Integer vector of length 7.
IPARAM(1) = Initialization flag.

IPARAM(2) = Number of good digits in the function.
Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.
Default: 100.

IPARAM(4) = Maximum number of function evaluations.
Default: 400.

IPARAM(5) = Maximum number of gradient evaluations.
Default: 400.

IPARAM(6) = Hessian initialization parameter.
If IPARAM(6) = 0, the Hessian is initialized to the identity matrix; otherwise, it is initialized to a diagonal matrix containing

$$\max\left(\left|f\left(t\right)\right|, f_s\right) * s_i^2$$

on the diagonal where t = XGUESS, fs = FSCALE, and s = XSCALE.
Default: 0.

IPARAM(7) = Maximum number of Hessian evaluations.
Default: Not used in BCONG.

RPARAM — Real vector of length 7.

RPARAM(1) = Scaled gradient tolerance.

The *i*-th component of the scaled gradient at x is calculated as

$$\frac{|g_i| * \max(|x_i|, 1/s_i)}{\max(|f(x)|, f_s)}$$

where $g = \nabla f(x)$, s = XSCALE, and f_s = FSCALE.

Default:

$$\sqrt{\varepsilon}, \sqrt[3]{\varepsilon}$$

in double where ε is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)

The *i*-th component of the scaled step between two points *x* and *y* is computed as

$$\frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)}$$

where *s* = XSCALE.

Default: $\varepsilon^{2/3}$ where ε is the machine precision.

RPARAM(3) = Relative function tolerance.

Default: $\max(10^{-10}, \varepsilon^{2/3})$, $\max(10^{-20}, \varepsilon^{2/3})$ in double where ε is the machine precision.

RPARAM(4) = Absolute function tolerance.

Default: Not used in BCONG.

RPARAM(5) = False convergence tolerance.

Default: 100ε where ε is the machine precision.

RPARAM(6) = Maximum allowable step size.

Default: 1000 $\max(\varepsilon_1, \varepsilon_2)$ where

$$\varepsilon_1 \sqrt{\sum_{i=1}^{n} (s_i t_i)^2}$$

$\varepsilon_2 = \| s \|_2$, *s* = XSCALE, and *t* = XGUESS.

RPARAM(7) = Size of initial trust region radius.

Default: based on the initial scaled Cauchy step.

If double precision is required, then DU4INF is called and RPARAM is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to “Error Handling” in the Introduction.

Description

The routine BCONG uses a quasi-Newton method and an active set strategy to solve minimization problems subject to simple bounds on the variables. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

$$\text{subject to } \; l \le x \le u$$

From a given starting point x^c, an active set IA, which contains the indices of the variables at their bounds, is built. A variable is called a “free variable” if it is not in the active set. The routine then computes the search direction for the free variables according to the formula

$$d = -B^{-1} g^c$$

where B is a positive definite approximation of the Hessian and g^c is the gradient evaluated at x^c; both are computed with respect to the free variables. The search direction for the variables in IA is set to zero. A line search is used to find a new point x^n ,

$$x^n = x^c + \lambda d, \lambda \in (0, 1]$$

such that

$$f(x^n) \le f(x^c) + \alpha g^T d, \qquad \alpha \in (0, 0.5)$$

Finally, the optimality conditions

$$\|g(x_i)\| \le \varepsilon, l_i < x_i < u_i$$

$$g(x_i) < 0, x_i = u_i$$

$$g(x_i) > 0, x_i = l_i$$

are checked, where ε is a gradient tolerance. When optimality is not achieved, B is updated according to the BFGS formula:

$$B \leftarrow B - \frac{Bss^T B}{s^T Bs} + \frac{yy^T}{y^T s}$$

where $s = x^n - x^c$ and $y = g^n - g^c$. Another search direction is then computed to begin the next iteration.

The active set is changed only when a free variable hits its bounds during an iteration or the optimality condition is met for the free variables but not for all variables in IA, the active set. In the latter case, a variable that violates the optimality condition will be dropped out of IA. For more details on the quasi-Newton method and line search, see Dennis and Schnabel (1983). For more detailed information on active set strategy, see Gill and Murray (1976).

BCODH

Minimizes a function of N variables subject to bounds on the variables using a modified Newton method and a finite-difference Hessian.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function to be minimized. The usage is CALL FCN (N, X, F), where

N – Length of X. (Input)

X – Vector of length N at which point the function is evaluated. (Input)
X should not be changed by FCN.

F – The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

GRAD — User-supplied SUBROUTINE to compute the gradient at the point X. The usage is CALL GRAD (N, X, G), where

N – Length of X and G. (Input)

X – Vector of length N at which point the gradient is evaluated. (Input)
X should not be changed by GRAD.

G – The gradient evaluated at the point X. (Output)

GRAD must be declared EXTERNAL in the calling program.

IBTYPE — Scalar indicating the types of bounds on variables. (Input)

IBTYPE	**Action**
0	User will supply all the bounds.
1	All variables are nonnegative.
2	All variables are nonpositive.
3	User supplies only the bounds on 1st variable, all other variables will have the same bounds.

XLB — Vector of length N containing the lower bounds on the variables. (Input)

XUB — Vector of length N containing the upper bounds on the variables. (Input)

X — Vector of length N containing the computed solution. (Output)

Optional Arguments

N — Dimension of the problem. (Input)
Default: N = size (X,1).

XGUESS — Vector of length N containing the initial guess of the minimum. (Input)
Default: XGUESS = 0.0.

XSCALE — Vector of length N containing the diagonal scaling matrix for the variables. (Input)
XSCALE is used mainly in scaling the gradient and the distance between two points. In the absence of other information, set all entries to 1.0.
Default: XSCALE = 1.0.

FSCALE — Scalar containing the function scaling. (Input)
FSCALE is used mainly in scaling the gradient. In the absence of other information, set FSCALE to 1.0.
Default: FSCALE = 1.0.

IPARAM — Parameter vector of length 7. (Input/Output)
Set IPARAM(1) to zero for default values of IPARAM and RPARAM. See Comment 4.
Default: IPARAM = 0.

RPARAM — Parameter vector of length 7. (Input/Output)
See Comment 4.

FVALUE — Scalar containing the value of the function at the computed solution. (Output)

FORTRAN 90 Interface

Generic: `CALL BCODH (FCN, GRAD, IBTYPE, XLB, XUB, X [,…])`

Specific: The specific interface names are S_BCODH and D_BCODH.

FORTRAN 77 Interface

Single:
```
CALL BCODH (FCN, GRAD, N, XGUESS, IBTYPE, XLB, XUB, XSCALE,
FSCALE, IPARAM, RPARAM, X, FVALUE)
```

Double: The double precision name is DBCODH.

Example

The problem

$$\min f(x) = 100\left(x_2 - x_1^2\right)^2 + \left(1 - x_1\right)^2$$
$$\text{subject to} \quad -2 \le x_1 \le 0.5$$
$$-1 \le x_2 \le 2$$

is solved with an initial guess (−1.2, 1.0), and default values for parameters.

```
      USE BCODH_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=2)
!
      INTEGER    IP, IPARAM(7), L, NOUT
      REAL       F, X(N), XGUESS(N), XLB(N), XUB(N)
      EXTERNAL   ROSBRK, ROSGRD
!
      DATA XGUESS/-1.2E0, 1.0E0/
      DATA XLB/-2.0E0, -1.0E0/, XUB/0.5E0, 2.0E0/
!
      IPARAM(1) = 0
      IP        = 0
!                                 Minimize Rosenbrock function using
!                                 initial guesses of -1.2 and 1.0
      CALL BCODH (ROSBRK, ROSGRD, IP, XLB, XUB, X, XGUESS=XGUESS, &
                 IPARAM=IPARAM, FVALUE=F)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) X, F, (IPARAM(L),L=3,5)
!
99999 FORMAT (’  The solution is ’, 6X, 2F8.3, //, ’  The function ’, &
            ’value is ’, F8.3, //, ’  The number of iterations is ’, &
            10X, I3, /, ’  The number of function evaluations is ’, &
            I3, /, ’  The number of gradient evaluations is ’, I3)
!
      END
!
      SUBROUTINE ROSBRK (N, X, F)
      INTEGER    N
      REAL       X(N), F
!
      F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
!
      RETURN
      END
      SUBROUTINE ROSGRD (N, X, G)
      INTEGER    N
      REAL       X(N), G(N)
!
      G(1) = -4.0E2*(X(2)-X(1)*X(1))*X(1) - 2.0E0*(1.0E0-X(1))
      G(2) = 2.0E2*(X(2)-X(1)*X(1))
!
      RETURN
      END
```

Output

```
The solution is          0.500   0.250

The function value is    0.250

The number of iterations is            17
The number of function evaluations is  26
The number of gradient evaluations is  18
```

Comments

1. Workspace may be explicitly provided, if desired, by use of B2ODH/DB2ODH. The reference is:

   ```
   CALL B2ODH (FCN, GRAD, N, XGUESS, IBTYPE, XLB, XUB, XSCALE,
   FSCALE, IPARAM, RPARAM, X, FVALUE, WK, IWK)
   ```

   The additional arguments are as follows:

   WK — Real work vector of length N * (N + 8). WK contains the following information on output: The second N locations contain the last step taken. The third N locations contain the last Newton step. The fourth N locations contain an estimate of the gradient at the solution. The final N^2 locations contain the Hessian at the approximate solution.

   IWK — Integer work vector of length N.

2. Informational errors

Type	Code	
3	1	Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.
4	2	The iterates appear to be converging to a noncritical point.
4	3	Maximum number of iterations exceeded.
4	4	Maximum number of function evaluations exceeded.
4	5	Maximum number of gradient evaluations exceeded.
4	6	Five consecutive steps have been taken with the maximum step length.
2	7	Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or STEPTL is too big.
4	7	Maximum number of Hessian evaluations exceeded.

3. The first stopping criterion for BCODH occurs when the norm of the gradient is less than the given gradient tolerance (RPARAM(1)). The second stopping criterion for BCODH occurs when the scaled distance between the last two steps is less than the step tolerance (RPARAM(2)).

4. If the default parameters are desired for BCODH, then set IPARAM(1) to zero and call the routine BCODH. Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM; then the following steps should be taken before calling BCODH:

CALL U4INF (IPARAM, RPARAM)
Set nondefault values for desired IPARAM, RPARAM elements.

Note that the call to U4INF will set IPARAM and RPARAM to their default values so only nondefault values need to be set above.

The following is a list of the parameters and the default values:

IPARAM — Integer vector of length 7.
IPARAM(1) = Initialization flag.

IPARAM(2) = Number of good digits in the function.
Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.
Default: 100.

IPARAM(4) = Maximum number of function evaluations.
Default: 400.

IPARAM(5) = Maximum number of gradient evaluations.
Default: 400.

IPARAM(6) = Hessian initialization parameter.
Default: Not used in BCODH.

IPARAM(7) = Maximum number of Hessian evaluations.
Default: 100.

RPARAM — Real vector of length 7.
RPARAM(1) = Scaled gradient tolerance.
The *i*-th component of the scaled gradient at x is calculated as

$$\frac{|g_i| * \max\left(|x_i|, 1/s_i\right)}{\max\left(\left|f\left(x\right)\right|, f_s\right)}$$

where $g = \nabla f(x)$, s = XSCALE, and f_s = FSCALE.
Default:

$$\sqrt{\varepsilon}, \sqrt[3]{\varepsilon}$$

in double where ε is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)
The *i*-th component of the scaled step between two points *x* and *y* is computed as

$$\frac{|x_i - y_i|}{\max\left(|x_i|, 1/s_i\right)}$$

where s = XSCALE.
Default: $\varepsilon^{2/3}$ where ε is the machine precision.

RPARAM(3) = Relative function tolerance.
Default: $\max(10^{-10}, \varepsilon^{2/3})$, $\max(10^{-20}, \varepsilon^{2/3})$ in double where ε is the machine precision.

RPARAM(4) = Absolute function tolerance.
Default: Not used in BCODH.

RPARAM(5) = False convergence tolerance.
Default: 100ε where ε is the machine precision.

RPARAM(6) = Maximum allowable step size.
Default: 1000 $\max(\varepsilon_1, \varepsilon_2)$ where

$$\varepsilon_1 \sqrt{\sum\nolimits_{i=1}^{n} \left(s_i t_i\right)^2}$$

$\varepsilon_2 = \| s \|_2$, s = XSCALE, and t = XGUESS.

RPARAM(7) = Size of initial trust region radius.
Default: based on the initial scaled Cauchy step.

If double precision is required, then DU4INF is called and RPARAM is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to "Error Handling" in the Introduction.

Description

The routine BCODH uses a modified Newton method and an active set strategy to solve minimization problems subject to simple bounds on the variables. The problem is stated as

$$\min_{x \in \mathbf{R}^n} f(x)$$

$$\text{subject to } l \le x \le u$$

From a given starting point x^c, an active set IA, which contains the indices of the variables at their bounds, is built. A variable is called a "free variable" if it is not in the active set. The routine then computes the search direction for the free variables according to the formula

$$d = -H^{-1} g^c$$

where H is the Hessian and g^c is the gradient evaluated at x^c; both are computed with respect to the free variables. The search direction for the variables in IA is set to zero. A line search is used to find a new point x^n ,

$$x^n = x^c + \lambda d, \lambda \in (0, 1]$$

such that

$$f(x^n) \le f(x^c) + \alpha g^T d, \qquad \alpha \in (0, 0.5)$$

Finally, the optimality conditions

$$\|g(x_i)\| \le \varepsilon,\ l_i < x_i < u_i$$

$$g(x_i) < 0,\ x_i = u_i$$

$$g(x_i) > 0,\ x_i = l_i$$

are checked where ε is a gradient tolerance. When optimality is not achieved, another search direction is computed to begin the next iteration. This process is repeated until the optimality criterion is met.

The active set is changed only when a free variable hits its bounds during an iteration or the optimality condition is met for the free variables but not for all variables in IA, the active set. In the latter case, a variable that violates the optimality condition will be dropped out of IA. For more details on the modified Newton method and line search, see Dennis and Schnabel (1983). For more detailed information on active set strategy, see Gill and Murray (1976).

Since a finite-difference method is used to estimate the Hessian for some single precision calculations, an inaccurate estimate of the Hessian may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended. Also, whenever the exact Hessian can be easily provided, routine BCOAH (page 1263) should be used instead.

BCOAH

Minimizes a function of N variables subject to bounds on the variables using a modified Newton method and a user-supplied Hessian.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function to be minimized. The usage is CALL FCN (N, X, F), where

N – Length of X. (Input)

X – Vector of length N at which point the function is evaluated. (Input)
X should not be changed by FCN.

F – The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

GRAD — User-supplied `SUBROUTINE` to compute the gradient at the point `X`. The usage is `CALL GRAD (N, X, G)`, where

`N` – Length of `X` and `G`. (Input)

`X` – Vector of length `N` at which point the gradient is evaluated. (Input)
`X` should not be changed by `GRAD`.

`G` – The gradient evaluated at the point `X`. (Output)

`GRAD` must be declared `EXTERNAL` in the calling program.

HESS — User-supplied `SUBROUTINE` to compute the Hessian at the point `X`. The usage is `CALL HESS (N, X, H, LDH)`, where

`N` – Length of `X`. (Input)

`X` – Vector of length `N` at which point the Hessian is evaluated. (Input)
`X` should not be changed by `HESS`.

`H` – The Hessian evaluated at the point `X`. (Output)

`LDH` – Leading dimension `of H` exactly as specified in the dimension statement of the calling program. (Input)

`HESS` must be declared `EXTERNAL` in the calling program.

IBTYPE — Scalar indicating the types of bounds on variables. (Input)

`IBTYPE`	**Action**
0	User will supply all the bounds.
1	All variables are nonnegative.
2	All variables are nonpositive.
3	User supplies only the bounds on 1st variable, all other variables will have the same bounds.

XLB — Vector of length `N` containing the lower bounds on the variables. (Input)

XUB — Vector of length `N` containing the upper bounds on the variables. (Input)

X — Vector of length `N` containing the computed solution. (Output)

Optional Arguments

N — Dimension of the problem.   (Input)
Default: `N` = size (`X`,1).

XGUESS — Vector of length `N` containing the initial guess.   (Input)
Default: `XGUESS` = 0.0.

XSCALE — Vector of length `N` containing the diagonal scaling matrix for the variables.   (Input)
`XSCALE` is used mainly in scaling the gradient and the distance between two points. In the absence of other information, set all entries to 1.0.
Default: `XSCALE` = 1.0.

FSCALE — Scalar containing the function scaling.   (Input)
`FSCALE` is used mainly in scaling the gradient. In the absence of other information, set `FSCALE` to 1.0.
Default: `FSCALE` = 1.0.

IPARAM — Parameter vector of length 7.   (Input/Output)
Set `IPARAM`(1) to zero for default values of `IPARAM` and `RPARAM`. See Comment 4.
Default: `IPARAM` = 0.

RPARAM — Parameter vector of length 7.   (Input/Output)
See Comment 4.

FVALUE — Scalar containing the value of the function at the computed solution.   (Output)

FORTRAN 90 Interface

Generic: `CALL BCOAH (FCN, GRAD, HESS, IBTYPE, XLB, XUB, X [,…])`

Specific: The specific interface names are `S_BCOAH` and `D_BCOAH`.

FORTRAN 77 Interface

Single:
```
CALL BCOAH (FCN, GRAD, HESS, N, XGUESS, IBTYPE, XLB, XUB,
XSCALE, FSCALE, IPARAM, RPARAM, X, FVALUE)
```

Double: The double precision name is `DBCOAH`.

Example

The problem

$$\min f(x) = 100\left(x_2 - x_1^2\right)^2 + \left(1 - x_1\right)^2$$
$$\text{subject to} \quad -2 \le x_1 \le 0.5$$
$$-1 \le x_2 \le 2$$

is solved with an initial guess (−1.2, 1.0), and default values for parameters.

```
      USE BCOAH_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=2)
!
      INTEGER    IP, IPARAM(7), L, NOUT
      REAL       F, X(N), XGUESS(N), XLB(N), XUB(N)
      EXTERNAL   ROSBRK, ROSGRD, ROSHES
!
      DATA XGUESS/-1.2E0, 1.0E0/
      DATA XLB/-2.0E0, -1.0E0/, XUB/0.5E0, 2.0E0/
!
      IPARAM(1) = 0
      IP        = 0
!                                 Minimize Rosenbrock function using
!                                 initial guesses of -1.2 and 1.0
      CALL BCOAH (ROSBRK, ROSGRD, ROSHES, IP, XLB, XUB, X, &
                 XGUESS=XGUESS,IPARAM=IPARAM, FVALUE=F)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) X, F, (IPARAM(L),L=3,5), IPARAM(7)
!
99999 FORMAT ('  The solution is ', 6X, 2F8.3, //, '  The function ', &
            'value is ', F8.3, //, '  The number of iterations is ', &
            10X, I3, /, '  The number of function evaluations is ', &
            I3, /, '  The number of gradient evaluations is ', I3, /, &
            '  The number of Hessian evaluations is  ', I3)
!
      END
!
      SUBROUTINE ROSBRK (N, X, F)
      INTEGER    N
      REAL       X(N), F
!
      F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
!
      RETURN
      END
!
      SUBROUTINE ROSGRD (N, X, G)
      INTEGER    N
      REAL       X(N), G(N)
!
      G(1) = -4.0E2*(X(2)-X(1)*X(1))*X(1) - 2.0E0*(1.0E0-X(1))
      G(2) = 2.0E2*(X(2)-X(1)*X(1))
!
      RETURN
      END
```

```
!
      SUBROUTINE ROSHES (N, X, H, LDH)
      INTEGER    N, LDH
      REAL       X(N), H(LDH,N)
!
      H(1,1) = -4.0E2*X(2) + 1.2E3*X(1)*X(1) + 2.0E0
      H(2,1) = -4.0E2*X(1)
      H(1,2) = H(2,1)
      H(2,2) = 2.0E2
!
      RETURN
      END
```

Output

```
The solution is          0.500   0.250

The function value is    0.250

The number of iterations is            18
The number of function evaluations is  29
The number of gradient evaluations is  19
The number of Hessian evaluations is   18
```

Comments

1. Workspace may be explicitly provided, if desired, by use of B2OAH/DB2OAH. The reference is:

   ```
   CALL B2OAH (FCN, GRAD, HESS, N, XGUESS, IBTYPE, XLB,
          XUB, XSCALE, FSCALE, IPARAM, RPARAM, X,
          FVALUE, WK, IWK)
   ```

   The additional arguments are as follows:

   WK — Work vector of length N * (N + 8). WK contains the following information on output: The second N locations contain the last step taken. The third N locations contain the last Newton step. The fourth N locations contain an estimate of the gradient at the solution. The final N^2 locations contain the Hessian at the approximate solution.

   IWK — Work vector of length N.

2. Informational errors

Type	Code	
3	1	Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.
4	2	The iterates appear to be converging to a noncritical point.

4	3	Maximum number of iterations exceeded.
4	4	Maximum number of function evaluations exceeded.
4	5	Maximum number of gradient evaluations exceeded.
4	6	Five consecutive steps have been taken with the maximum step length.
2	7	Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or STEPTL is too big.
4	7	Maximum number of Hessian evaluations exceeded.
3	8	The last global step failed to locate a lower point than the current X value.

3. The first stopping criterion for BCOAH occurs when the norm of the gradient is less than the given gradient tolerance (RPARAM(1)). The second stopping criterion for BCOAH occurs when the scaled distance between the last two steps is less than the step tolerance (RPARAM(2)).

4. If the default parameters are desired for BCOAH, then set IPARAM(1) to zero and call the routine BCOAH. Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM, then the following steps should be taken before calling BCOAH:

   CALL U4INF (IPARAM, RPARAM)
   Set nondefault values for desired IPARAM, RPARAM elements.

   Note that the call to U4INF will set IPARAM and RPARAM to their default values so only nondefault values need to be set above.

   The following is a list of the parameters and the default values:

   IPARAM — Integer vector of length 7.
   IPARAM(1) = Initialization flag.

   IPARAM(2) = Number of good digits in the function.
   Default: Machine dependent.

   IPARAM(3) = Maximum number of iterations.
   Default: 100.

   IPARAM(4) = Maximum number of function evaluations.
   Default: 400.

   IPARAM(5) = Maximum number of gradient evaluations.
   Default: 400.

   IPARAM(6) = Hessian initialization parameter.
   Default: Not used in BCOAH.

IPARAM(7) = Maximum number of Hessian evaluations.
Default: 100.

RPARAM — Real vector of length 7.
RPARAM(1) = Scaled gradient tolerance.
The *i*-th component of the scaled gradient at x is calculated as

$$\frac{|g_i| * \max(|x_i|, 1/s_i)}{\max(|f(x)|, f_s)}$$

where $g = \nabla f(x)$, s = XSCALE, and f_s = FSCALE.
Default:

$$\sqrt{\varepsilon}, \sqrt[3]{\varepsilon}$$

in double where ε is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)
The *i*-th component of the scaled step between two points *x* and *y* is computed as

$$\frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)}$$

where s = XSCALE.
Default: $\varepsilon^{2/3}$ where ε is the machine precision.

RPARAM(3) = Relative function tolerance.
Default: $\max(10^{-10}, \varepsilon^{2/3})$, $\max(10^{-20}, \varepsilon^{2/3})$ in double where ε is the machine precision.

RPARAM(4) = Absolute function tolerance.
Default: Not used in BCOAH.

RPARAM(5) = False convergence tolerance.
Default: 100ε where ε is the machine precision.

RPARAM(6) = Maximum allowable step size.
Default: 1000 $\max(\varepsilon_1, \varepsilon_2)$ where

$$\varepsilon_1 \sqrt{\sum_{i=1}^{n} (s_i t_i)^2}$$

$\varepsilon_2 = \| s \|_2$, s = XSCALE, and t = XGUESS.

RPARAM(7) = Size of initial trust region radius.
Default: based on the initial scaled Cauchy step.

If double precision is required, then DU4INF is called and RPARAM is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to “Error Handling” in the Introduction.

Description

The routine BCOAH uses a modified Newton method and an active set strategy to solve minimization problems subject to simple bounds on the variables. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

$$\text{subject to } l \le x \le u$$

From a given starting point x^c, an active set IA, which contains the indices of the variables at their bounds, is built. A variable is called a “free variable” if it is not in the active set. The routine then computes the search direction for the free variables according to the formula

$$d = -H^{-1} g^c$$

where H is the Hessian and g^c is the gradient evaluated at x^c; both are computed with respect to the free variables. The search direction for the variables in IA is set to zero. A line search is used to find a new point x^n ,

$$x^n = x^c + \lambda d, \lambda \in (0, 1]$$

such that

$$f(x^n) \le f(x^c) + \alpha g^T d, \ \alpha \in (0, 0.5)$$

Finally, the optimality conditions

$$\|g(x_i)\| \le \varepsilon, \ l_i < x_i < u_i$$

$$g(x_i) < 0, \ x_i = u_i$$

$$g(x_i) > 0, \ x_i = l_i$$

are checked where ε is a gradient tolerance. When optimality is not achieved, another search direction is computed to begin the next iteration. This process is repeated until the optimality criterion is met.

The active set is changed only when a free variable hits its bounds during an iteration or the optimality condition is met for the free variables but not for all variables in IA, the active set. In the latter case, a variable that violates the optimality condition will be dropped out of IA. For more details on the modified Newton method and line search, see Dennis and Schnabel (1983). For more detailed information on active set strategy, see Gill and Murray (1976).

BCPOL

Minimizes a function of N variables subject to bounds on the variables using a direct search complex algorithm.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function to be minimized. The usage is CALL FCN (N, X, F), where

N – Length of X. (Input)

X – Vector of length N at which point the function is evaluated. (Input)
X should not be changed by FCN.

F – The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

IBTYPE — Scalar indicating the types of bounds on variables. (Input)

IBTYPE	**Action**
0	User will supply all the bounds.
1	All variables are nonnegative.
2	All variables are nonpositive.
3	User supplies only the bounds on the first, variable. All other variables will have the same bounds.

XLB — Vector of length N containing the lower bounds on the variables. (Input, if IBTYPE = 0; output, if IBTYPE = 1 or 2; input/output, if IBTYPE = 3)

XUB — Vector of length N containing the upper bounds on the variables. (Input, if IBTYPE = 0; output, if IBTYPE = 1 or 2; input/output, if IBTYPE = 3)

X — Real vector of length N containing the best estimate of the minimum found. (Output)

Optional Arguments

N — The number of variables. (Input)
Default: N = size (XGUESS,1).

XGUESS — Real vector of length N that contains an initial guess to the minimum. (Input)
Default: XGUESS = 0.0.

FTOL — First convergence criterion. (Input)
The algorithm stops when a relative error in the function values is less than FTOL, i.e. when (F(worst) – F(best)) < FTOL * (1 + ABS(F(best))) where F(worst) and F(best) are the function values of the current worst and best point, respectively. Second convergence criterion. The algorithm stops when the standard deviation of the function values at the 2 * N current points is less than FTOL. If the subroutine terminates prematurely, try again with a smaller value FTOL.
Default: FTOL = 1.0e-4 for single and 1.0d-8 for double precision.

MAXFCN — On input, maximum allowed number of function evaluations. (Input/ Output)
On output, actual number of function evaluations needed.
Default: MAXFCN = 300.

FVALUE — Function value at the computed solution. (Output)

FORTRAN 90 Interface

Generic: CALL BCPOL (FCN, IBTYPE, XLB, XUB, X [,…])

Specific: The specific interface names are S_BCPOL and D_BCPOL.

FORTRAN 77 Interface

Single: CALL BCPOL (FCN, N, XGUESS, IBTYPE, XLB, XUB, FTOL, MAXFCN, X, FVALUE)

Double: The double precision name is DBCPOL.

Example

The problem

$$\min f(x) = 100\left(x_2 - x_1^2\right)^2 + \left(1 - x_1\right)^2$$
$$\text{subject to} \quad -2 \le x_1 \le 0.5$$
$$-1 \le x_2 \le 2$$

is solved with an initial guess (–1.2, 1.0), and the solution is printed.

```
      USE BCPOL_INT
      USE UMACH_INT
!                                 Variable declarations
      INTEGER    N
      PARAMETER  (N=2)
!
      INTEGER    IBTYPE, K, NOUT
      REAL       FTOL, FVALUE, X(N), XGUESS(N), XLB(N), XUB(N)
      EXTERNAL   FCN
```

```
!
!                                  Initializations
!                                  XGUESS = (-1.2,  1.0)
!                                  XLB    = (-2.0, -1.0)
!                                  XUB    = ( 0.5,  2.0)
      DATA  XGUESS/-1.2, 1.0/, XLB/-2.0E0, -1.0E0/, XUB/0.5E0, 2.0E0/
!
      FTOL   = 1.0E-5
      IBTYPE = 0
!
      CALL BCPOL (FCN, IBTYPE, XLB, XUB, X, XGUESS=XGUESS, FTOL=FTOL, &
                 FVALUE=FVALUE)
!
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) (X(K),K=1,N), FVALUE
99999 FORMAT ('  The best estimate for the minimum value of the', /, &
            '  function is X = (', 2(2X,F4.2), ')', /, '  with ', &
            'function value FVALUE = ', E12.6)
!
      END
!                                  External function to be minimized
      SUBROUTINE FCN (N, X, F)
      INTEGER    N
      REAL       X(N), F
!
      F = 100.0*(X(2)-X(1)*X(1))**2 + (1.0-X(1))**2
      RETURN
      END
```

Output

```
The best estimate for the minimum value of the
function is X = (  0.50  0.25)
with function value FVALUE = 0.250002E+00
```

Comments

1. Workspace may be explicitly provided, if desired, by use of B2POL/DB2POL. The reference is:

   ```
   CALL B2POL (FCN, N, XGUESS, IBTYPE, XLB, XUB, FTOL,
          MAXFCN, X, FVALUE, WK)
   ```

   The additional argument is:

   WK — Real work vector of length 2 * N**2 + 5 * N

2. Informational error

Type	Code	
3	1	The maximum number of function evaluations is exceeded.

3. Since BCPOL uses only function-value information at each step to determine a new approximate minimum, it could be quite inefficient on smooth problems compared to other methods such as those implemented in routine BCONF (page 1243), which takes

into account derivative information at each iteration. Hence, routine `BCPOL` should only be used as a last resort. Briefly, a set of 2 * `N` points in an `N`-dimensional space is called a complex. The minimization process iterates by replacing the point with the largest function value by a new point with a smaller function value. The iteration continues until all the points cluster sufficiently close to a minimum.

Description

The routine `BCPOL` uses the complex method to find a minimum point of a function of n variables. The method is based on function comparison; no smoothness is assumed. It starts with $2n$ points $x_1, x_2, \ldots, x_{2n}$. At each iteration, a new point is generated to replace the worst point x_j, which has the largest function value among these $2n$ points. The new point is constructed by the following formula:

$$x_k = c + \alpha(c - x_j)$$

where

$$c = \frac{1}{2n-1} \sum_{i \neq j} x_i$$

and α ($\alpha > 0$) is the *reflection coefficient*.

When x_k is a best point, that is, when $f(x_k) \le f(x_i)$ for $i = 1, \ldots, 2n$, an expansion point is computed $x_e = c + \beta(x_k - c)$, where $\beta(\beta > 1)$ is called the *expansion coefficient*. If the new point is a worst point, then the complex would be contracted to get a better new point. If the contraction step is unsuccessful, the complex is shrunk by moving the vertices halfway toward the current best point. Whenever the new point generated is beyond the bound, it will be set to the bound. This procedure is repeated until one of the following stopping criteria is satisfied:

Criterion 1:

$$f_{best} - f_{worst} \le \varepsilon_f (1. + |f_{best}|)$$

Criterion 2:

$$\sum_{i=1}^{2n} (f_i - \frac{\sum_{j=1}^{2n} f_j}{2n})^2 \le \varepsilon_f$$

where $f_i = f(x_i)$, $f_j = f(x_j)$, and ε_f is a given tolerance. For a complete description, see Nelder and Mead (1965) or Gill et al. (1981).

BCLSF

Solves a nonlinear least squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm and a finite-difference Jacobian.

Required Arguments

FCN — User-supplied `SUBROUTINE` to evaluate the function to be minimized. The usage is `CALL FCN (M, N, X, F)`, where

`M` – Length of `F`. (Input)

`N` – Length of `X`. (Input)

`X` – The point at which the function is evaluated. (Input)
`X` should not be changed by `FCN`.

`F` – The computed function at the point `X`. (Output)

`FCN` must be declared `EXTERNAL` in the calling program.

M — Number of functions. (Input)

IBTYPE — Scalar indicating the types of bounds on variables. (Input)

IBTYPE	**Action**
0	User will supply all the bounds.
1	All variables are nonnegative.
2	All variables are nonpositive.
3	User supplies only the bounds on 1st variable, all other variables will have the same bounds.

XLB — Vector of length `N` containing the lower bounds on variables. (Input, if `IBTYPE` = 0; output, if `IBTYPE` = 1 or 2; input/output, if `IBTYPE` = 3)

XUB — Vector of length `N` containing the upper bounds on variables. (Input, if `IBTYPE` = 0; output, if `IBTYPE` = 1 or 2; input/output, if `IBTYPE` = 3)

X — Vector of length `N` containing the approximate solution. (Output)

Optional Arguments

N — Number of variables. (Input)
`N` must be less than or equal to `M`.
Default: `N` = size (`X`,1).

XGUESS — Vector of length `N` containing the initial guess. (Input)
Default: `XGUESS` = 0.0.

XSCALE — Vector of length `N` containing the diagonal scaling matrix for the variables. (Input)
`XSCALE` is used mainly in scaling the gradient and the distance between two points. By default, the values for `XSCALE` are set internally. See `IPARAM`(6) in Comment 4.

FSCALE — Vector of length `M` containing the diagonal scaling matrix for the functions. (Input)
`FSCALE` is used mainly in scaling the gradient. In the absence of other information, set all entries to 1.0.
Default: `FSCALE` = 1.0.

IPARAM — Parameter vector of length 6. (Input/Output)
Set `IPARAM`(1) to zero for default values of `IPARAM` and `RPARAM`. See Comment 4.
Default: `IPARAM`= 0.

RPARAM — Parameter vector of length 7. (Input/Output)
See Comment 4.

FVEC — Vector of length `M` containing the residuals at the approximate solution. (Output)

FJAC — `M` by `N` matrix containing a finite difference approximate Jacobian at the approximate solution. (Output)

LDFJAC — Leading dimension of `FJAC` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDFJAC` = size (`FJAC` ,1).

FORTRAN 90 Interface

Generic: `CALL BCLSF (FCN, M, IBTYPE, XLB, XUB, X [,…])`

Specific: The specific interface names are `S_BCLSF` and `D_BCLSF`.

FORTRAN 77 Interface

Single:

```
CALL BCLSF (FCN, M, N, XGUESS, IBTYPE, XLB, XUB, XSCALE,
FSCALE, IPARAM, RPARAM, X, FVEC, FJAC, LDFJAC)
```

Double: The double precision name is `DBCLSF`.

Example

The nonlinear least squares problem

$$\min_{x\in\mathbf{R}^2} \frac{1}{2}\sum_{i=1}^{2} f_i\left(x\right)^2$$

$$\text{subject to } -2 \le x_1 \le 0.5$$

$$-1 \le x_2 \le 2$$

where

$$f_1(x) = 10\left(x_2 - x_1^2\right) \text{ and } f_2(x) = \left(1 - x_1\right)$$

is solved with an initial guess (−1.2, 1.0) and default values for parameters.

```
      USE BCLSF_INT
      USE UMACH_INT
!                                 Declaration of variables
      INTEGER    M, N
      PARAMETER  (M=2, N=2)
!
      INTEGER    IPARAM(7), ITP, NOUT
      REAL       FSCALE(M), FVEC(M), X(N), XGUESS(N), XLB(N), XS(N), XUB(N)
      EXTERNAL   ROSBCK
!                                 Compute the least squares for the
!                                 Rosenbrock function.
      DATA XGUESS/-1.2E0, 1.0E0/
      DATA XLB/-2.0E0, -1.0E0/, XUB/0.5E0, 2.0E0/
!                                 All the bounds are provided
      ITP = 0
!                                 Default parameters are used
      IPARAM(1) = 0
!
      CALL BCLSF (ROSBCK, M, ITP, XLB, XUB, X, XGUESS=XGUESS, &
                 IPARAM=IPARAM, FVEC=FVEC)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) X, FVEC, IPARAM(3), IPARAM(4)
!
99999 FORMAT ('  The solution is ', 2F9.4, //, '  The function ', &
            'evaluated at the solution is ', /, 18X, 2F9.4, //, &
            '  The number of iterations is ', 10X, I3, /, '  The ', &
            'number of function evaluations is ', I3, /)
      END
!
      SUBROUTINE ROSBCK (M, N, X, F)
      INTEGER    M, N
      REAL       X(N), F(M)
!
      F(1) = 1.0E1*(X(2)-X(1)*X(1))
      F(2) = 1.0E0 - X(1)
      RETURN
      END
```

Output

```
The solution is    0.5000   0.2500

The function evaluated at the solution is
0.0000   0.5000

The number of iterations is            15
The number of function evaluations is  20
```

Comments

1. Workspace may be explicitly provided, if desired, by use of B2LSF/DB2LSF. The reference is:

   ```
   CALL B2LSF (FCN, M, N, XGUESS, IBTYPE, XLB, XUB, XSCALE, FSCALE,
   IPARAM, RPARAM, X, FVEC, FJAC, LDFJAC, WK, IWK)
   ```

   The additional arguments are as follows:

   WK — Work vector of length 11 * N + 3 * M – 1. WK contains the following information on output: The second N locations contain the last step taken. The third N locations contain the last Gauss-Newton step. The fourth N locations contain an estimate of the gradient at the solution.

   IWK — Work vector of length 2 * N containing the permutations used in the QR factorization of the Jacobian at the solution.

2. Informational errors

Type	Code	
3	1	Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.
3	2	The iterates appear to be converging to a noncritical point.
4	3	Maximum number of iterations exceeded.
4	4	Maximum number of function evaluations exceeded.
3	6	Five consecutive steps have been taken with the maximum step length.
2	7	Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or STEPTL is too big.

3. The first stopping criterion for BCLSF occurs when the norm of the function is less than the absolute function tolerance. The second stopping criterion occurs when the norm of the scaled gradient is less than the given gradient tolerance. The third stopping criterion for BCLSF occurs when the scaled distance between the last two steps is less than the step tolerance.

4. If the default parameters are desired for BCLSF, then set IPARAM(1) to zero and call the routine BCLSF. Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM, then the following steps should be taken before calling BCLSF:

   ```
   CALL U4LSF (IPARAM, RPARAM)
   ```
   Set nondefault values for desired IPARAM, RPARAM elements.

   Note that the call to U4LSF will set IPARAM and RPARAM to their default values so only nondefault values need to be set above.

   The following is a list of the parameters and the default values:

IPARAM — Integer vector of length 6.
IPARAM(1) = Initialization flag.

IPARAM(2) = Number of good digits in the function.
Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.
Default: 100.

IPARAM(4) = Maximum number of function evaluations.
Default: 400.

IPARAM(5) = Maximum number of Jacobian evaluations.
Default: 100.

IPARAM(6) = Internal variable scaling flag.
If IPARAM(6) = 1, then the values for XSCALE are set internally.
Default: 1.

RPARAM — Real vector of length 7.
RPARAM(1) = Scaled gradient tolerance.
The *i*-th component of the scaled gradient at x is calculated as

$$\frac{|g_i| * \max\left(|x_i|, 1/s_i\right)}{\left\|F(x)\right\|_2^2}$$

where

$$g_i = \left(J(x)^T F(x)\right)_i * \left(f_s\right)_i^2$$

$J(x)$ is the Jacobian, s = XSCALE, and f_s = FSCALE.
Default:

$$\sqrt{\varepsilon}, \sqrt[3]{\varepsilon}$$

in double where ε is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)
The i-th component of the scaled step between two points *x* and *y* is computed as

$$\frac{|x_i - y_i|}{\max\left(|x_i|, 1/s_i\right)}$$

where s = XSCALE.

Default: $\varepsilon^{2/3}$ where ε is the machine precision.

`RPARAM`(3) = Relative function tolerance.
Default: max($10^{-10,}$ $\varepsilon^{2/3),}$ max(10^{-20}, $\varepsilon^{2/3}$) in double where ε is the machine precision.

`RPARAM`(4) = Absolute function tolerance.
Default: max (10^{-20}, ε^2), max(10^{-40}, ε^2) in double where ε is the machine precision.

`RPARAM`(5) = False convergence tolerance.
Default: 100 ε where ε is the machine precision.

`RPARAM`(6) = Maximum allowable step size.
Default: 1000 max(ε_1, ε_2) where

$$\varepsilon_1 = \sqrt{\sum\nolimits_{i=1}^{n} \left(s_i t_i\right)^2}$$

$\varepsilon_2 = \|s\|_2$, s = `XSCALE`, and t = `XGUESS`.

`RPARAM`(7) = Size of initial trust region radius.
Default: based on the initial scaled Cauchy step.

If double precision is desired, then `DU4LSF` is called and `RPARAM` is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to "Error Handling" in the Introduction.

Description

The routine `BCLSF` uses a modified Levenberg-Marquardt method and an active set strategy to solve nonlinear least squares problems subject to simple bounds on the variables. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} \frac{1}{2} F(x)^T F(x) = \frac{1}{2} \sum_{i=1}^{m} f_i(x)^2$$

$$\text{subject to } l \le x \le u$$

where $m \ge n$, $F : \mathbf{R}^n \rightarrow \mathbf{R}^m$, and $f_i(x)$ is the i-th component function of $F(x)$. From a given starting point, an active set IA, which contains the indices of the variables at their bounds, is built. A variable is called a "free variable" if it is not in the active set. The routine then computes the search direction for the free variables according to the formula

$$d = -\left(J^T J + \mu I\right)^{-1} J^T F$$

where μ is the Levenberg-Marquardt parameter, $F = F(x)$, and J is the Jacobian with respect to the free variables. The search direction for the variables in IA is set to zero. The trust region approach discussed by Dennis and Schnabel (1983) is used to find the new point. Finally, the optimality conditions are checked. The conditions are

$$\|g(x_i)\| \le \varepsilon, \; l_i < x_i < u_i$$

$$g(x_i) < 0, \;\; x_i = u_i$$

$$g(x_i) > 0, \; x_i = l_i$$

where ε is a gradient tolerance. This process is repeated until the optimality criterion is achieved.

The active set is changed only when a free variable hits its bounds during an iteration or the optimality condition is met for the free variables but not for all variables in IA, the active set. In the latter case, a variable that violates the optimality condition will be dropped out of IA. For more detail on the Levenberg-Marquardt method, see Levenberg (1944), or Marquardt (1963). For more detailed information on active set strategy, see Gill and Murray (1976).

Since a finite-difference method is used to estimate the Jacobian for some single precision calculations, an inaccurate estimate of the Jacobian may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended. Also, whenever the exact Jacobian can be easily provided, routine BCLSJ (page 1281) should be used instead.

BCLSJ

Solves a nonlinear least squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm and a user-supplied Jacobian.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function to be minimized. The usage is CALL FCN (M, N, X, F), where

M – Length of F. (Input)

N – Length of X. (Input)

X – The point at which the function is evaluated. (Input)
X should not be changed by FCN.

F – The computed function at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

JAC — User-supplied SUBROUTINE to evaluate the Jacobian at a point X. The usage is CALL JAC (M, N, X, FJAC, LDFJAC), where

M – Length of F. (Input)

N – Length of X. (Input)

X – The point at which the function is evaluated. (Input)
X should not be changed by FCN.

FJAC – The computed M by N Jacobian at the point X. (Output)

LDFJAC – Leading dimension of FJAC. (Input)

JAC must be declared EXTERNAL in the calling program.

M — Number of functions. (Input)

IBTYPE — Scalar indicating the types of bounds on variables. (Input)

IBTYPE	**Action**
0	User will supply all the bounds.
1	All variables are nonnegative.
2	All variables are nonpositive.
3	User supplies only the bounds on 1st variable, all other variables will have the same bounds.

XLB — Vector of length N containing the lower bounds on variables. (Input, if IBTYPE = 0; output, if IBTYPE = 1 or 2; input/output, if IBTYPE = 3)

XUB — Vector of length N containing the upper bounds on variables. (Input, if IBTYPE = 0; output, if IBTYPE = 1 or 2; input/output, if IBTYPE = 3)

X — Vector of length N containing the approximate solution. (Output)

Optional Arguments

N — Number of variables. (Input)
N must be less than or equal to M.
Default: N = size (X,1).

XGUESS — Vector of length N containing the initial guess. (Input)
Default: XGUESS = 0.0.

XSCALE — Vector of length N containing the diagonal scaling matrix for the variables. (Input)
XSCALE is used mainly in scaling the gradient and the distance between two points. By default, the values for XSCALE are set internally. See IPARAM(6) in Comment 4.

FSCALE — Vector of length M containing the diagonal scaling matrix for the functions. (Input)

FSCALE is used mainly in scaling the gradient. In the absence of other information, set all entries to 1.0.

Default: FSCALE = 1.0.

IPARAM — Parameter vector of length 6. (Input/Output)

Set IPARAM(1) to zero for default values of IPARAM and RPARAM. See Comment 4.

Default: IPARAM= 0.

RPARAM — Parameter vector of length 7. (Input/Output)

See Comment 4.

FVEC — Vector of length M containing the residuals at the approximate solution. (Output)

FJAC — M by N matrix containing a finite difference approximate Jacobian at the approximate solution. (Output)

LDFJAC — Leading dimension of FJAC exactly as specified in the dimension statement of the calling program. (Input)

Default: LDFJAC size = (FJAC,1).

FORTRAN 90 Interface

Generic: `CALL BCLSJ (FCN, JAC, M, IBTYPE, XLB, XUB, X [,…])`

Specific: The specific interface names are S_BCLSJ and D_BCLSJ.

FORTRAN 77 Interface

Single:

```
CALL BCLSJ (FCN, JAC, M, N, XGUESS, IBTYPE, XLB, XUB,
XSCALE, FSCALE, IPARAM, RPARAM, X, FVEC, FJAC,
LDFJAC)
```

Double: The double precision name is DBCLSJ.

Example

The nonlinear least squares problem

$$\min_{x\in\mathbf{R}^2} \frac{1}{2}\sum_{i=1}^{2} f_i\left(x\right)^2$$

$$\text{subject to } -2 \le x_1 \le 0.5$$

$$-1 \le x_2 \le 2$$

where

$$f_1(x) = 10\left(x_2 - x_1^2\right) \text{ and } f_2(x) = \left(1 - x_1\right)$$

is solved with an initial guess (−1.2, 1.0) and default values for parameters.

```
      USE BCLSJ_INT
      USE UMACH_INT
!                                 Declaration of variables
      INTEGER    LDFJAC, M, N
      PARAMETER  (LDFJAC=2, M=2, N=2)
!
      INTEGER    IPARAM(7), ITP, NOUT
      REAL       FVEC(M), RPARAM(7), X(N), XGUESS(N), XLB(N), XUB(N)
      EXTERNAL   ROSBCK, ROSJAC
!                                 Compute the least squares for the
!                                 Rosenbrock function.
      DATA XGUESS/-1.2E0, 1.0E0/
      DATA XLB/-2.0E0, -1.0E0/, XUB/0.5E0, 2.0E0/
!                                 All the bounds are provided
      ITP = 0
!                                 Default parameters are used
      IPARAM(1) = 0
!
      CALL BCLSJ (ROSBCK,ROSJAC,M,ITP,XLB,XUB,X,XGUESS=XGUESS, &
                 IPARAM=IPARAM, FVEC=FVEC)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) X, FVEC, IPARAM(3), IPARAM(4)
!
99999 FORMAT ('  The solution is ', 2F9.4, //, '  The function ', &
            'evaluated at the solution is ', /, 18X, 2F9.4, //, &
            '  The number of iterations is ', 10X, I3, /, '  The ', &
            'number of function evaluations is ', I3, /)
      END
!
      SUBROUTINE ROSBCK (M, N, X, F)
      INTEGER    M, N
      REAL       X(N), F(M)
!
      F(1) = 1.0E1*(X(2)-X(1)*X(1))
      F(2) = 1.0E0 - X(1)
      RETURN
      END
!
      SUBROUTINE ROSJAC (M, N, X, FJAC, LDFJAC)
      INTEGER    M, N, LDFJAC
      REAL       X(N), FJAC(LDFJAC,N)
!
      FJAC(1,1) = -20.0E0*X(1)
      FJAC(2,1) = -1.0E0
      FJAC(1,2) = 10.0E0
      FJAC(2,2) = 0.0E0
      RETURN
      END
```

Output

```
The solution is    0.5000   0.2500

The function evaluated at the solution is
0.0000   0.5000

The number of iterations is            13
The number of function evaluations is  21
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `B2LSJ/DB2LSJ`. The reference is:

   ```
   CALL B2LSJ (FCN, JAC, M, N, XGUESS, IBTYPE, XLB, XUB, XSCALE,
   FSCALE, IPARAM, RPARAM, X, FVEC, FJAC, LDFJAC, WK, IWK)
   ```

   The additional arguments are as follows:

   WK — Work vector of length 11 * `N` + 3 * `M` – 1. `WK` contains the following information on output: The second `N` locations contain the last step taken. The third `N` locations contain the last Gauss-Newton step. The fourth `N` locations contain an estimate of the gradient at the solution.

   IWK — Work vector of length 2 * `N` containing the permutations used in the `QR` factorization of the Jacobian at the solution.

2. Informational errors

Type	Code	
3	1	Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.
3	2	The iterates appear to be converging to a noncritical point.
4	3	Maximum number of iterations exceeded.
4	4	Maximum number of function evaluations exceeded.
3	6	Five consecutive steps have been taken with the maximum step length.
4	5	Maximum number of Jacobian evaluations exceeded.
2	7	Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or `STEPTL` is too big.

3. The first stopping criterion for `BCLSJ` occurs when the norm of the function is less than the absolute function tolerance. The second stopping criterion occurs when the norm of the scaled gradient is less than the given gradient tolerance. The third stopping criterion for `BCLSJ` occurs when the scaled distance between the last two steps is less than the step tolerance.

4. If the default parameters are desired for `BCLSJ`, then set `IPARAM`(1) to zero and call the routine `BCLSJ`. Otherwise, if any nondefault parameters are desired for `IPARAM` or `RPARAM`, then the following steps should be taken before calling `BCLSJ`:

CALL U4LSF (IPARAM, RPARAM)
Set nondefault values for desired IPARAM, RPARAM elements.

Note that the call to U4LSF will set IPARAM and RPARAM to their default values so only nondefault values need to be set above.

The following is a list of the parameters and the default values:

IPARAM — Integer vector of length 6.
IPARAM(1) = Initialization flag.

IPARAM(2) = Number of good digits in the function.
Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.
Default: 100.

IPARAM(4) = Maximum number of function evaluations.
Default: 400.

IPARAM(5) = Maximum number of Jacobian evaluations.
Default: 100.

IPARAM(6) = Internal variable scaling flag.

If IPARAM(6) = 1, then the values for XSCALE are set internally.
Default: 1.

RPARAM — Real vector of length 7.
RPARAM(1) = Scaled gradient tolerance.
The *i*-th component of the scaled gradient at x is calculated as

$$\frac{|g_i| * \max\left(|x_i|, 1/s_i\right)}{\left\|F(x)\right\|_2^2}$$

where

$$g_i = \left(J(x)^T F(x)\right)_i * \left(f_s\right)_i^2$$

$J(x)$ is the Jacobian, s = XSCALE, and f_s = FSCALE.
Default:

$$\sqrt{\varepsilon}, \sqrt[3]{\varepsilon}$$

in double where ε is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)
The i-th component of the scaled step
between two points x and y is computed as

$$\frac{|x_i - y_i|}{\max\left(|x_i|, 1/s_i\right)}$$

where s = XSCALE.

Default: $\varepsilon^{2/3}$ where ε is the machine precision.

RPARAM(3) = Relative function tolerance.
Default: $\max(10^{-10}, \varepsilon^{2/3})$, $\max(10^{-20}, \varepsilon^{2/3})$ in double where ε is the machine precision.

RPARAM(4) = Absolute function tolerance.
Default: $\max(10^{-20}, \varepsilon^{2})$, $\max(10^{-40}, \varepsilon^{2})$ in double where ε is the machine precision.

RPARAM(5) = False convergence tolerance.
Default: 100ε where ε is the machine precision.

RPARAM(6) = Maximum allowable step size.
Default: 1000 $\max(\varepsilon_1, \varepsilon_2)$ where

$$\varepsilon_1 = \sqrt{\sum\nolimits_{i=1}^{n} \left(s_i t_i\right)^2}$$

$\varepsilon_2 = \|s\|_2$, s = XSCALE, and t = XGUESS.

RPARAM(7) = Size of initial trust region radius.
Default: based on the initial scaled Cauchy step.

If double precision is desired, then DU4LSF is called and RPARAM is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to ERROR HANDLING in the Introduction.

Description

The routine BCLSJ uses a modified Levenberg-Marquardt method and an active set strategy to solve nonlinear least squares problems subject to simple bounds on the variables. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} \frac{1}{2} F(x)^T F(x) = \frac{1}{2} \sum_{i=1}^{m} f_i(x)^2$$

subject to $l \le x \le u$

where $m \geq n$, $F : \mathbf{R}^n \rightarrow \mathbf{R}^m$, and $f_i(x)$ is the i-th component function of $F(x)$. From a given starting point, an active set IA, which contains the indices of the variables at their bounds, is built. A variable is called a “free variable” if it is not in the active set. The routine then computes the search direction for the free variables according to the formula

$$d = -(J^T J + \mu I)^{-1} J^T F$$

where is the Levenberg-Marquardt parameter, $F = F(x)$, and J is the Jacobian with respect to the free variables. The search direction for the variables in IA is set to zero. The trust region approach discussed by Dennis and Schnabel (1983) is used to find the new point. Finally, the optimality conditions are checked. The conditions are

$$\|g(x_i)\| \leq \varepsilon,\ l_i < x_i < u_i$$

$$g(x_i) < 0,\ \ x_i = u_i$$

$$g(x_i) > 0,\ x_i = l_i$$

where ε is a gradient tolerance. This process is repeated until the optimality criterion is achieved.

The active set is changed only when a free variable hits its bounds during an iteration or the optimality condition is met for the free variables but not for all variables in IA, the active set. In the latter case, a variable that violates the optimality condition will be dropped out of IA. For more detail on the Levenberg-Marquardt method, see Levenberg (1944) or Marquardt (1963). For more detailed information on active set strategy, see Gill and Murray (1976).

BCNLS

Solves a nonlinear least-squares problem subject to bounds on the variables and general linear constraints.

Required Arguments

FCN — User-supplied `SUBROUTINE` to evaluate the function to be minimized. The usage is
`CALL FCN (M, N, X, F)`, where
`M` – Number of functions. (Input)
`N` – Number of variables. (Input)
`X` – Array of length `N` containing the point at which the function will be evaluated. (Input)
`F` – Array of length `M` containing the computed function at the point `X`. (Output)
The routine `FCN` must be declared `EXTERNAL` in the calling program.

M — Number of functions. (Input)

C — `MCON` × `N` matrix containing the coefficients of the `MCON` general linear constraints. (Input)

BL — Vector of length `MCON` containing the lower limit of the general constraints. (Input).

BU — Vector of length MCON containing the upper limit of the general constraints. (Input).

IRTYPE — Vector of length MCON indicating the types of general constraints in the matrix C. (Input)
Let R(I) = C(I, 1)*X(1) + … + C(I, N)*X(N). Then the value of IRTYPE(I) signifies the following:

IRTYPE(I)	**I-th CONSTRAINT**
0	BL(I).EQ.R(I).EQ.BU(I)
1	R(I).LE.BU(I)
2	R(I).GE.BL(I)
3	BL(I).LE.R(I).LE.BU(I)

XLB — Vector of length N containing the lower bounds on variables; if there is no lower bound on a variable, then 1.0E30 should be set as the lower bound. (Input)

XUB — Vector of length N containing the upper bounds on variables; if there is no upper bound on a variable, then −1.0E30 should be set as the upper bound. (Input)

X — Vector of length N containing the approximate solution. (Output)

Optional Arguments

N — Number of variables. (Input)
Default: N = size (C,2).

MCON — The number of general linear constraints for the system, not including simple bounds. (Input)
Default: MCON = size (C,1).

LDC — Leading dimension of C exactly as specified in the dimension statement of the calling program. (Input)
LDC must be at least MCON.
Default: LDC = size (C,1).

XGUESS — Vector of length N containing the initial guess. (Input)
Default: XGUESS = 0.0.

RNORM — The Euclidean length of components of the function $f(x)$ after the approximate solution has been found. (Output).

ISTAT — Scalar indicating further information about the approximate solution X. (Output)
See the Comments section for a description of the tolerances and the vectors IPARAM and RPARAM.

ISTAT **Meaning**

1 The function $f(x)$ has a length less than TOLF = RPARAM(1). This is the expected value for ISTAT when an actual zero value of $f(x)$ is anticipated.

2 The function $f(x)$ has reached a local minimum. This is the expected value for ISTAT when a nonzero value of $f(x)$ is anticipated.

3 A small change (absolute) was noted for the vector x. A full model problem step was taken. The condition for ISTAT = 2 may also be satisfied, so that a minimum has been found. However, this test is made before the test for ISTAT = 2.

4 A small change (relative) was noted for the vector x. A full model problem step was taken. The condition for ISTAT = 2 may also be satisfied, so that a minimum has been found. However, this test is made before the test for ISTAT = 2.

5 The number of terms in the quadratic model is being restricted by the amount of storage allowed for that purpose. It is suggested, but not required, that additional storage be given for the quadratic model parameters. This is accessed through the vector
IPARAM, documented below.

6 Return for evaluation of function and Jacobian if reverse communication is desired. See the Comments below.

FORTRAN 90 Interface

Generic: `CALL BCNLS (FCN, M, C, BL, BU, IRTYPE, XLB, XUB, X [,…])`

Specific: The specific interface names are `S_BCNLS` and `D_BCNLS`.

FORTRAN 77 Interface

Single:
```
CALL BCNLS (FCN, M, N, MCON, C, LDC, BL, BU, IRTYPE,
XLB, XUB, XGUESS, X, RNORM, ISTAT)
```

Double: The double precision name is `DBCNLS`.

Example 1

This example finds the four variables x_1, x_2, x_3, x_4 that are in the model function

$$h(t) = x_1 e^{x_2 t} + x_3 e^{x_4 t}$$

There are values of *h(t)* at five values of *t*.

$h(0.05) = 2.206$

$h(0.1) = 1.994$

```
h(0.4) = 1.35
h(0.5) = 1.216
h(1.0) = 0.7358
```

There are also the constraints that $x_2, x_4 \le 0, x_1, x_3 \ge 0$, and x_2 and x_4 must be separated by at least 0.05. Nothing more about the values of the parameters is known so the initial guess is 0.

```
      USE BCNLS_INT
      USE UMACH_INT
      USE WRRRN_INT
      INTEGER    MCON, N
      PARAMETER  (MCON=1, N=4)
!                                 SPECIFICATIONS FOR PARAMETERS
      INTEGER    LDC, M
      PARAMETER  (M=5, LDC=MCON)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    IRTYPE(MCON), NOUT
      REAL       BL(MCON), C(MCON,N), RNORM, X(N), XLB(N), &
                XUB(N)
!                                 SPECIFICATIONS FOR SUBROUTINES
!                                 SPECIFICATIONS FOR FUNCTIONS
      EXTERNAL   FCN
!
      CALL UMACH (2, NOUT)
!                                 Define the separation between x(2)
!                                 and x(4)
      C(1,1) = 0.0
      C(1,2) = 1.0
      C(1,3) = 0.0
      C(1,4) = -1.0
      BL(1) = 0.05
      IRTYPE(1) = 2
!                                 Set lower bounds on variables
      XLB(1) = 0.0
      XLB(2) = 1.0E30
      XLB(3) = 0.0
      XLB(4) = 1.0E30
!                                 Set upper bounds on variables
      XUB(1) = -1.0E30
      XUB(2) = 0.0
      XUB(3) = -1.0E30
      XUB(4) = 0.0
!
      CALL BCNLS (FCN, M, C, BL, BL, IRTYPE, XLB, XUB, X, RNORM=RNORM)
      CALL WRRRN ('X', X, 1, N, 1)
      WRITE (NOUT,99999) RNORM
99999 FORMAT (/, 'rnorm = ', E10.5)
      END
!
      SUBROUTINE FCN (M, N, X, F)
!                                 SPECIFICATIONS FOR ARGUMENTS
      INTEGER    M, N
      REAL       X(*), F(*)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
```

```
      INTEGER    I
!                                  SPECIFICATIONS FOR SAVE VARIABLES
      REAL       H(5), T(5)
      SAVE       H, T
!                                  SPECIFICATIONS FOR INTRINSICS
      INTRINSIC  EXP
      REAL       EXP
!
      DATA T/0.05, 0.1, 0.4, 0.5, 1.0/
      DATA H/2.206, 1.994, 1.35, 1.216, 0.7358/
!
      DO 10  I=1, M
         F(I) = X(1)*EXP(X(2)*T(I)) + X(3)*EXP(X(4)*T(I)) - H(I)
   10 CONTINUE
      RETURN
      END
```

Output

```
                   X
       1       2       3       4
   1.999  -1.000   0.500  -9.954
rnorm = .42425E-03
```

Comments

1. Workspace may be explicitly provided, if desired, by use of B2NLS/DB2NLS. The reference is:

```
CALL B2NLS (FCN, M, N, MCON, C, LDC, BL, BU, IRTYPE, XLB, XUB,
XGUESS, X, RNORM,ISTAT, IPARAM, RPARAM, JAC, F, FJ, LDFJ,
IWORK, LIWORK, WORK, LWORK)
```

The additional arguments are as follows:

IPARAM — Integer vector of length six used to change certain default attributes of BCNLS. (Input).
If the default parameters are desired for BCNLS, set IPARAM(1) to zero. Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM, the following steps should be taken before calling B2NLS:

CALL B7NLS (IPARAM, RPARAM)
Set nondefault values for IPARAM and RPARAM.

If double precision is being used, DB7NLS should be called instead. Following is a list of parameters and the default values.

IPARAM(1) = Initialization flag.

IPARAM(2) = ITMAX, the maximum number of iterations allowed.
Default: 75

IPARAM(3) = a flag that suppresses the use of the quadratic model in the inner loop. If set to one, then the quadratic model is never used. Otherwise use the quadratic model where appropriate. This option decreases the amount of workspace as well as the computing overhead required. A user may wish to determine if the application really requires the use of the quadratic model.
Default: 0

IPARAM(4) = NTERMS, one more than the maximum number of terms used in the quadratic model.
Default: 5

IPARAM(5) = RCSTAT, a flag that determines whether forward or reverse communication is used. If set to zero, forward communication through functions FCN and JAC is used. If set to one, reverse communication is used, and the dummy routines B10LS/DB10LS and B11LS/DB11LS may be used in place of FCN and JAC, respectively. When BCNLS returns with ISTAT = 6, arrays F and FJ are filled with $f(x)$ and the Jacobian of $f(x)$, respectively. BCNLS is then called again.
Default: 0

IPARAM(6) = a flag that determines whether the analytic Jacobian, as supplied in JAC, is used, or if a finite difference approximation is computed. If set to zero, JAC is not accessed and finite differences are used. If set to one, JAC is used to compute the Jacobian.
Default: 0

RPARAM — Real vector of length 7 used to change certain default attributes of BCNLS. (Input)

For the description of RPARAM, we make the following definitions:
FC current value of the length of $f(x)$
FB best value of length of $f(x)$
FL value of length of $f(x)$ at the previous step
PV predicted value of length of $f(x)$, after the step is taken, using the approximating model
ε machine epsilon = amach(4)

The conditions |FB − PV| ≤ TOLSNR*FB and |FC − PV| ≤ TOLP*FB and |FC − FL| ≤ TOLSNR*FB together with taking a full model step, must be satisfied before the condition ISTAT = 2 is returned. (Decreasing any of the values for TOLF, TOLD, TOLX, TOLSNR, or TOLP will likely increase the number of iterations required for convergence.)
RPARAM(1) = TOLF, tolerance used for stopping when FC ≤ TOLF.
$\text{Default}: \min(1.\text{E}-5, \sqrt{\varepsilon})$

RPARAM(2) = TOLX, tolerance for stopping when change to x values has length less than or equal to TOLX*length of x values.
$\text{Default}: \min(1.\text{E}-5, \sqrt{\varepsilon})$

RPARAM(3) = TOLD, tolerance for stopping when change to x values has length less than pr equal to TOLD.
$\text{Default}: \min(1.\text{E}-5, \sqrt{\varepsilon})$

RPARAM(4) = TOLSNR, tolerance used in stopping condition ISTAT = 2.
Default: 1.E–5

RPARAM(5) = TOLP, tolerance used in stopping condition ISTAT = 2.
Default: 1.E–5

RPARAM(6) = TOLUSE, tolerance used to avoid values of x in the quadratic model's interpolation of previous points. Decreasing this value may result in more terms being included in the quadratic model.
$\text{Default}: \sqrt{\varepsilon}$

RPARAM(7) = COND, largest condition number to allow when solving for the quadratic model coefficients. Increasing this value may result in more terms being included in the quadratic model.
Default: 30

JAC — User-supplied SUBROUTINE to evaluate the Jacobian. The usage is
CALL JAC(M, N, X, FJAC, LDFJAC), where
M – Number of functions. (Input)
N – Number of variables. (Input)
X – Array of length N containing the point at which the Jacobian will be evaluated. (Input)
FJAC – The computed M × N Jacobian at the point X. (Output)
LDFJAC – Leading dimension of the array FJAC. (Input)
The routine JAC must be declared EXTERNAL in the calling program.

F — Real vector of length N used to pass $f(x)$ if reverse communication (IPARAM(4)) is enabled. (Input)

FJ — Real array of size M × N used to store the Jacobian matrix of $f(x)$ if reverse communication (IPARAM(4)) is enabled. (Input)
Specifically,

$$FJ(i,j) = \frac{\partial f_i}{\partial x_j}$$

LDFJ — Leading dimension of FJ exactly as specified in the dimension statement of the calling program. (Input)

IWORK — Integer work vector of length LIWORK.

LIWORK — Length of work vector IWORK. LIWORK must be at least
5MCON + 12N + 47 + MAX(M, N)

WORK — Real work vector of length LWORK

LWORK — Length of work vector WORK. LWORK must be at least 41N + 6M + 11MCON + (M + MCON)(N + 1) + NA(NA + 7) + 8 MAX(M, N) + 99. Where NA = MCON + 2N + 6.

2. Informational errors

Type	Code	
3	1	The function *f* (*x*) has reached a value that may be a local minimum. However, the bounds on the trust region defining the size of the step are being hit at each step. Thus, the situation is suspect. (Situations of this type can occur when the solution is at infinity at some of the components of the unknowns, *x*).
3	2	The model problem solver has noted a value for the linear or quadratic model problem residual vector length that is greater than or equal to the current value of the function, i.e. the Euclidean length of *f* (*x*). This situation probably means that the evaluation of *f (x)* has more uncertainty or noise than is possible to account for in the tolerances used to not a local minimum. The value of *x* is suspect, but a minimum has probably been found.
3	3	More than ITMAX iterations were taken to obtain the solution. The value obtained for *x* is suspect, although it is the best set of *x* values that occurred in the entire computation. The value of ITMAX can be increased though the IPARAM vector.

Description

The routine BCNLS solves the nonlinear least squares problem

$$\min \sum_{i=1}^{m} f_i\left(x\right)^2$$

subject to

$$b_l \le Cx \le b_u$$
$$x_l \le x \le x_u$$

BCNLS is based on the routine DQED by R.J. Hanson and F.T. Krogh. The section of BCNLS that approximates, using finite differences, the Jacobian of *f(x)* is a modification of JACBF by D.E. Salane.

Example 2

This example solves the same problem as the last example, but reverse communication is used to evaluate *f*(*x*) and the Jacobian of *f*(*x*). The use of the quadratic model is turned off.

```
      USE B2NLS_INT
      USE UMACH_INT
      USE WRRRN_INT
      INTEGER    LDC, LDFJ, M, MCON, N
      PARAMETER  (M=5, MCON=1, N=4, LDC=MCON, LDFJ=M)
!                                  Specifications for local variables
      INTEGER   I, IPARAM(6), IRTYPE(MCON), ISTAT, IWORK(1000), &
                LIWORK, LWORK, NOUT
      REAL       BL(MCON), C(MCON,N), F(M), FJ(M,N), RNORM, RPARAM(7), &
                WORK(1000), X(N), XGUESS(N), XLB(N), XUB(N)
      REAL       H(5), T(5)
      SAVE       H, T
      INTRINSIC  EXP
      REAL       EXP
!                                  Specifications for subroutines
      EXTERNAL   B7NLS
!                                  Specifications for functions
      EXTERNAL   B10LS, B11LS
!
      DATA T/0.05, 0.1, 0.4, 0.5, 1.0/
      DATA H/2.206, 1.994, 1.35, 1.216, 0.7358/
!
      CALL UMACH (2, NOUT)
!                                  Define the separation between x(2)
!                                  and x(4)
      C(1,1)    = 0.0
      C(1,2)    = 1.0
      C(1,3)    = 0.0
      C(1,4)    = -1.0
      BL(1)     = 0.05
      IRTYPE(1) = 2
!                                  Set lower bounds on variables
      XLB(1) = 0.0
      XLB(2) = 1.0E30
      XLB(3) = 0.0
      XLB(4) = 1.0E30
!                                  Set upper bounds on variables
      XUB(1) = -1.0E30
      XUB(2) = 0.0
      XUB(3) = -1.0E30
      XUB(4) = 0.0
!                                  Set initial guess to 0.0
      XGUESS = 0.0E0
!                                  Call B7NLS to set default parameters
      CALL B7NLS (IPARAM, RPARAM)
!                                  Suppress the use of the quadratic
!                                  model, evaluate functions and
!                                  Jacobian by reverse communication
      IPARAM(3) = 1
      IPARAM(5) = 1
      IPARAM(6) = 1
      LWORK     = 1000
      LIWORK    = 1000
!                                  Specify dummy routines for FCN
!                                  and JAC since we are using reverse
!                                  communication
   10 CONTINUE
      CALL B2NLS (B10LS, M, N, MCON, C, LDC, BL, BL, IRTYPE, XLB, &
                 XUB, XGUESS, X, RNORM, ISTAT, IPARAM, RPARAM, &
                 B11LS, F, FJ, LDFJ, IWORK, LIWORK, WORK, LWORK)
!
!                                  Evaluate functions if the routine
!                                  returns with ISTAT = 6
      IF (ISTAT .EQ. 6) THEN
         DO 20  I=1, M
            FJ(I,1) = EXP(X(2)*T(I))
            FJ(I,2) = T(I)*X(1)*FJ(I,1)
            FJ(I,3) = EXP(X(4)*T(I))
            FJ(I,4) = T(I)*X(3)*FJ(I,3)
            F(I) = X(1)*FJ(I,1) + X(3)*FJ(I,3) - H(I)
   20    CONTINUE
         GO TO 10
      END IF
!
      CALL WRRRN ('X', X, 1, N, 1)
      WRITE (NOUT,99999) RNORM
99999 FORMAT (/, 'rnorm = ', E10.5)
      END
```

Output

```
                     X
      1       2       3       4
   1.999  -1.000   0.500  -9.954
rnorm = .42413E-03
```

DLPRS

Solves a linear programming problem via the revised simplex algorithm.

Required Arguments

A — `M` by `NVAR` matrix containing the coefficients of the `M` constraints. (Input)

BL — Vector of length `M` containing the lower limit of the general constraints; if there is no lower limit on the `I`-th constraint, then `BL(I)` is not referenced. (Input)

BU — Vector of length `M` containing the upper limit of the general constraints; if there is no upper limit on the `I`-th constraint, then `BU(I)` is not referenced; if there are no range constraints, `BL` and `BU` can share the same storage locations. (Input)

C — Vector of length NVAR containing the coefficients of the objective function. (Input)

IRTYPE — Vector of length M indicating the types of general constraints in the matrix A. (Input)
Let R(I) = A(I, 1) * XSOL(1) + … + A(I, NVAR) * XSOL(NVAR). Then, the value of IRTYPE(I) signifies the following:

IRTYPE(I)	**I-th Constraint**
0	BL(I).EQ.R(I).EQ.BU(I)
1	R(I).LE.BU(I)
2	R(I).GE.BL(I)
3	BL(I).LE.R(I).LE.BU(I)

OBJ — Value of the objective function. (Output)

XSOL — Vector of length NVAR containing the primal solution. (Output)

DSOL — Vector of length M containing the dual solution. (Output)

Optional Arguments

M — Number of constraints. (Input)
Default: M = size (A,1).

NVAR — Number of variables. (Input)
Default: NVAR = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
LDA must be at least M.
Default: LDA = size (A,1).

XLB — Vector of length NVAR containing the lower bound on the variables; if there is no lower bound on a variable, then 1.0E30 should be set as the lower bound. (Input)
Default: XLB = 0.0.

XUB — Vector of length NVAR containing the upper bound on the variables; if there is no upper bound on a variable, then −1.0E30 should be set as the upper bound. (Input)
Default: XUB = 3.4e38 for single precision and 1.79d + 308 for double precision.

FORTRAN 90 Interface

Generic: CALL DLPRS (A, BL, BU, C, IRTYPE, OBJ, XSOL, DSOL [,…])

Specific: The specific interface names are S_DLPRS and D_DLPRS.

FORTRAN 77 Interface

Single: CALL DLPRS (M, NVAR, A, LDA, BL, BU, C, IRTYPE, XLB, XUB, OBJ, XSOL, DSOL)

Double: The double precision name is DDLPRS.

Example

A linear programming problem is solved.

```
      USE DLPRS_INT
      USE UMACH_INT
      USE SSCAL_INT
      INTEGER    LDA, M, NVAR
      PARAMETER  (M=2, NVAR=2, LDA=M)
!                                  M = number of constraints
!                                  NVAR = number of variables
!
      INTEGER    I, IRTYPE(M), NOUT
      REAL       A(LDA,NVAR), B(M), C(NVAR), DSOL(M), OBJ, XLB(NVAR), &
                XSOL(NVAR), XUB(NVAR)
!
!                                  Set values for the following problem
!
!                                  Max 1.0*XSOL(1) + 3.0*XSOL(2)
!
!                                  XSOL(1) + XSOL(2) .LE. 1.5
!                                  XSOL(1) + XSOL(2) .GE. 0.5
!
!                                  0 .LE. XSOL(1) .LE. 1
!                                  0 .LE. XSOL(2) .LE. 1
!
      DATA XLB/2*0.0/, XUB/2*1.0/
      DATA A/4*1.0/, B/1.5, .5/, C/1.0, 3.0/
      DATA IRTYPE/1, 2/
!                                  To maximize, C must be multiplied by
!                                  -1.
      CALL SSCAL (NVAR, -1.0E0, C, 1)
!                                  Solve the LP problem.  Since there is
!                                  no range constraint, only B is
!                                  needed.
      CALL DLPRS (A, B, B, C, IRTYPE, OBJ, XSOL, DSOL, &
                 XUB=XUB)
!                                  OBJ must be multiplied by -1 to get
!                                  the true maximum.
      OBJ = -OBJ
!                                  DSOL must be multiplied by -1 for
!                                  maximization.
      CALL SSCAL (M, -1.0E0, DSOL, 1)
!                                  Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) OBJ, (XSOL(I),I=1,NVAR), (DSOL(I),I=1,M)
!
99999 FORMAT (//, '   Objective       = ', F9.4, //, '   Primal ',&
             'Solution =', 2F9.4, //, '   Dual solution   =', 2F9.4)
!
      END
```

Output

```
Objective       =    3.5000

Primal Solution =   0.5000   1.0000

Dual solution   =   1.0000   0.0000
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `D2PRS/DD2PRS`. The reference is:

   ```
   CALL D2PRS (M, NVAR, A, LDA, BL, BU, C, IRTYPE, XLB, XUB, OBJ,
   XSOL, DSOL, AWK, LDAWK, WK, IWK)
   ```

   The additional arguments are as follows:

   AWK — Real work array of dimension 1 by 1. (`AWK` is not used in the new implementation of the revised simplex algorithm. It is retained merely for calling sequence consistency.)

   LDAWK — Leading dimension of `AWK` exactly as specified in the dimension statement of the calling program. `LDAWK` should be 1. (`LDAWK` is not used in the new implementation of the revised simplex algorithm. It is retained merely for calling sequence consistency.)

   WK — Real work vector of length `M` * (`M` + 28).

   IWK — Integer work vector of length 29 * `M` + 3 * `NVAR`.

2. Informational errors

Type	Code	
3	1	The problem is unbounded.
4	2	Maximum number of iterations exceeded.
3	3	The problem is infeasible.
4	4	Moved to a vertex that is poorly conditioned; using double precision may help.
4	5	The bounds are inconsistent.

Description

The routine DLPRS uses a revised simplex method to solve linear programming problems, i.e., problems of the form

$$\min_{x \in \mathbf{R}^n} c^T x$$

$$\text{subject to } b_l \le Ax \le b_u$$

$$x_l \le x \le x_u$$

where c is the objective coefficient vector, A is the coefficient matrix, and the vectors b_l, b_u, x_l and x_u are the lower and upper bounds on the constraints and the variables, respectively.

For a complete description of the revised simplex method, see Murtagh (1981) or Murty (1983).

SLPRS

Solves a sparse linear programming problem via the revised simplex algorithm.

Required Arguments

A — Vector of length NZ containing the coefficients of the M constraints. (Input)

IROW — Vector of length NZ containing the row numbers of the corresponding element in A. (Input)

JCOL — Vector of length NZ containing the column numbers of the corresponding elements in *A*. (Input)

BL — Vector of length M containing the lower limit of the general constraints; if there is no lower limit on the I-th constraint, then BL(I) is not referenced. (Input)

BU — Vector of length M containing the upper lower limit of the general constraints; if there is no upper limit on the I-th constraint, then BU(I) is not referenced. (Input)

C — Vector of length NVAR containing the coefficients of the objective function. (Input)

IRTYPE — Vector of length M indicating the types of general constraints in the matrix A. (Input)
Let R(I) = A(I, 1)*XSOL(1) + … + A(I, NVAR)*XSOL(NVAR)

IRTYPE(I)	**I-th CONSTRAINT**
0	BL(I) = R(I) = BU(I)
1	R(I) ≤ BU(I)
2	R(I) ≥ BL(I)
3	BL(I) ≤ R(I) ≤ BU(I)

OBJ — Value of the objective function. (Output)

XSOL — Vector of length `NVAR` containing the primal solution. (Output)

DSOL — Vector of length `M` containing the dual solution. (Output)

Optional Arguments

M — Number of constraints. (Input)
Default: `M` = size (`IRTYPE`,1).

NVAR — Number of variables. (Input)
Default: `NVAR` = size (`C`,1).

NZ — Number of nonzero coefficients in the matrix *A*. (Input)
Default: `NZ` = size (`A`,1).

XLB — Vector of length `NVAR` containing the lower bound on the variables; if there is no lower bound on a variable, then 1.0E30 should be set as the lower bound. (Input)
Default: `XLB` = 0.0.

XUB — Vector of length `NVAR` containing the upper bound on the variables; if there is no upper bound on a variable, then −1.0E30 should be set as the upper bound. (Input)
Default: `XLB` = 3.4e38 for single precision and 1.79d + 308 for double precision.

FORTRAN 90 Interface

Generic:

```
CALL SLPRS (A, IROW, JCOL, BL, BU, C, IRTYPE,
OBJ, XSOL, DSOL [,…])
```

Specific: The specific interface names are `S_SLPRS` and `D_SLPRS`.

FORTRAN 77 Interface

Single:

```
CALL SLPRS (M, NVAR, NZ, A, IROW, JCOL, BL, BU, C, IRTYPE,
XLB, XUB, OBJ, XSOL, DSOL)
```

Double: The double precision name is `DSLPRS`.

Example

Solve a linear programming problem, with

$$A = \begin{bmatrix} 0 & 0.5 & & & \\ & 1 & 0.5 & & \\ & & 1 & \ddots & \\ & & & \ddots & 0.5 \\ & & & & 1 \end{bmatrix}$$

defined in sparse coordinate format.

```
      USE SLPRS_INT
      USE UMACH_INT
      INTEGER    M, NVAR
      PARAMETER  (M=200, NVAR=200)
!                                  Specifications for local variables
      INTEGER    INDEX, IROW(3*M), J, JCOL(3*M), NOUT, NZ
      REAL       A(3*M), DSOL(M), OBJ, XSOL(NVAR)
      INTEGER    IRTYPE(M)
      REAL       B(M), C(NVAR), XL(NVAR), XU(NVAR)
!                                  Specifications for subroutines
      DATA B/199*1.7, 1.0/
      DATA C/-1.0, -2.0, -3.0, -4.0, -5.0, -6.0, -7.0, -8.0, -9.0, &
      -10.0, 190*-1.0/
      DATA XL/200*0.1/
      DATA XU/200*2.0/
      DATA IRTYPE/200*1/
!
      CALL UMACH (2, NOUT)
!                                  Define A
      INDEX = 1
      DO 10  J=2, M
!                                  Superdiagonal element
         IROW(INDEX) = J - 1
         JCOL(INDEX) = J
         A(INDEX)    = 0.5
!                                  Diagonal element
         IROW(INDEX+1) = J
         JCOL(INDEX+1) = J
         A(INDEX+1) = 1.0
         INDEX      = INDEX + 2
   10 CONTINUE
      NZ = INDEX - 1
!
!
      XL(4) = 0.2
      CALL SLPRS (A, IROW, JCOL, B, B, C, IRTYPE, OBJ, XSOL, DSOL, &
                  NZ=NZ, XLB=XL, XUB=XU)
!
      WRITE (NOUT,99999) OBJ
!
99999 FORMAT (/, 'The value of the objective function is ', E12.6)
!
      END
```

Output

```
The value of the objective function is -.280971E+03
```

Comments

Workspace may be explicitly provided, if desired, by use of S2PRS/DS2PRS. The reference is:

```
CALL S2PRS (M, NVAR, NZ, A, IROW, JCOL, BL, BU, C,
       IRTYPE, XLB, XUB, OBJ, XSOL, DSOL,
       IPARAM, RPARAM, COLSCL, ROWSCL, WORK,
       LW, IWORK, LIW)
```

The additional arguments are as follows:

IPARAM — Integer parameter vector of length 12. If the default parameters are desired for SLPRS, then set IPARAM(1) to zero and call the routine SLPRS. Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM, then the following steps should be taken before calling SLPRS:

CALL S5PRS (IPARAM, RPARAM)
Set nondefault values for IPARAM and RPARAM.

Note that the call to S5PRS will set IPARAM and RPARAM to their default values so only nondefault values need to be set above.

IPARAM(1) = 0 indicates that a minimization problem is solved. If set to 1, a maximization problem is solved.
Default: 0

IPARAM(2) = switch indicating the maximum number of iterations to be taken before returning to the user. If set to zero, the maximum number of iterations taken is set to 3*(NVARS+M). If positive, that value is used as the iteration limit.
Default: IPARAM(2) = 0

IPARAM(3) = indicator for choosing how columns are selected to enter the basis. If set to zero, the routine uses the steepest edge pricing strategy which is the best local move. If set to one, the minimum reduced cost pricing strategy is used. The steepest edge pricing strategy generally uses fewer iterations than the minimum reduced cost pricing, but each iteration costs more in terms of the amount of calculation performed. However, this is very problem-dependent.
Default: IPARAM(3) = 0

IPARAM(4) = MXITBR, the number of iterations between recalculating the error in the primal solution is used to monitor the error in solving the linear system. This is an expensive calculation and every tenth iteration is generally enough.
Default: IPARAM(4) = 10

IPARAM(5) = NPP, the number of negative reduced costs (at most) to be found at each iteration of choosing a variable to enter the basis. If set to zero, NPP = NVARS will be used, implying that all of the reduced costs are computed at each such step. This “Partial pricing” may increase the total number of iterations required. However, it decreases the number of calculation required at each iteration. The effect on overall efficiency is very problem-dependent. If set to some positive number, that value is used as NPP.
Default: IPARAM(5) = 0

IPARAM(6) = IREDFQ, the number of steps between basis matrix redecompositions. Redecompositions also occur whenever the linear systems for the primal and dual systems have lost half their working precision.
Default: IPARAM(6) = 50

IPARAM(7) = LAMAT, the length of the portion of WORK that is allocated to sparse matrix storage and decomposition. LAMAT must be greater than NZ + NVARS + 4.
Default: LAMAT = NZ + NVARS + 5

IPARAM(8) = LBM, then length of the portion of IWORK that is allocated to sparse matrix storage and decomposition. LBM must be positive.
Default: LBM = 8*M

IPARAM(9) = switch indicating that partial results should be saved after the maximum number of iterations, IPARAM(2), or at the optimum. If IPARAM(9) is not zero, data essential to continuing the calculation is saved to a file, attached to unit number IPARAM(9). The data saved includes all the information about the sparse matrix A and information about the current basis. If IPARAM(9) is set to zero, partial results are not saved. It is the responsibility of the calling program to open the output file.

IPARAM(10) = switch indicating that partial results have been computed and stored on unit number IPARAM(10), if greater than zero. If IPARAM(10) is zero, a new problem is started.
Default: IPARAM(10) = 0

IPARAM(11) = switch indicating that the user supplies scale factors for the columns of the matrix *A*. If IPARAM(11) = 0, SLPRS computes the scale factors as the reciprocals of the max norm of each column. If IPARAM(11) is set to one, element I of the vector COLSCL is used as the scale factor for column I of the matrix *A*. The scaling is implicit, so no input data is actually changed.
Default: IPARAM(11) = 0

IPARAM(12) = switch indicating that the user supplied scale factors for the rows of the matrix *A*. If IPARAM(12) is set to zero, no row scaling is one. If IPARAM(12) is set to 1, element I of the vector ROWSCL is used as the scale factor for row I of the matrix *A*. The scaling is implicit, so no input data is actually changed.
Default: IPARAM(12) = 0

RPARAM — Real parameter vector of length 7.

RPARAM(1) = COSTSC, a scale factor for the vector of costs. Normally SLPRS computes this scale factor to be the reciprocal of the max norm if the vector costs after the column scaling has been applied. If RPARAM(1) is zero, SLPRS compute COSTSC.
Default: RPARAM(1) = 0.0

RPARAM(2) = ASMALL, the smallest magnitude of nonzero entries in the matrix *A*. If RPARAM(2) is nonzero, checking is done to ensure that all elements of *A* are at least as

large as RPARAM(2). Otherwise, no checking is done.
Default: RPARAM(2) = 0.0

RPARAM(3) = ABIG, the largest magnitude of nonzero entries in the matrix *A*. If RPARAM(3) is nonzero, checking is done to ensure that all elements of *A* are no larger than RPARAM(3). Otherwise, no checking is done.
Default: RPARAM(3) = 0.0

RPARAM(4) = TOLLS, the relative tolerance used in checking if the residuals are feasible. RPARAM(4) is nonzero, that value is used as TOLLS, otherwise the default value is used.
Default: TOLLS = 1000.0*amach(4)

RPARAM(5) = PHI, the scaling factor used to scale the reduced cost error estimates. In some environments, it may be necessary to reset PHI to the range [0.01, 0.1], particularly on machines with short word length and working precision when solving a large problem. If RPARAM(5) is nonzero, that value is used as PHI, otherwise the default value is used.
Default: PHI = 1.0

RPARAM(6) = TOLABS, an absolute error test on feasibility. Normally a relative test is used with TOLLS (see RPARAM(4)). If this test fails, an absolute test will be applied using the value TOLABS.
Default: TOLABS = 0.0

RPARAM(7) = pivot tolerance of the underlying sparse factorization routine. If RPARAM(7) is set to zero, the default pivot tolerance is used, otherwise, the RPARAM(7) is used.
Default: RPARAM(7) = 0.1

COLSCL — Array of length NVARS containing column scale factors for the matrix *A*. (Input).
COLSCL is not used if IPARAM(11) is set to zero.

ROWSCL — Array of length M containing row scale factors for the matrix *A*. (Input)
ROWSCL is not used if IPARAM(12) is set to zero.

WORK — Work array of length LW.

LW — Length of real work array. LW must be at least
2 + 2NZ + 9NVAR + 27M + MAX(NZ + NVAR + 8, 4NVAR + 7).

IWORK — Integer work array of length LIW.

LIW — Length of integer work array. LIW must be at least
1 + 3NVAR + 41M + MAX(NZ + NVAR + 8, 4NVAR + 7).

Description

This subroutine solves problems of the form

$$\min c^T x$$

subject to

$$b_l \le Ax \le b_u,$$
$$x_l \le x \le x_u$$

where *c* is the objective coefficient vector, *A* is the coefficient matrix, and the vectors b_l, b_u, x_l, and x_u are the lower and upper bounds on the constraints and the variables, respectively. `SLPRS` is designed to take advantage of sparsity in *A*. The routine is based on `DPLO` by Hanson and Hiebert.

QPROG

Solves a quadratic programming problem subject to linear equality/inequality constraints.

Required Arguments

NEQ — The number of linear equality constraints. (Input)

A — `NCON` by `NVAR` matrix. (Input)
The matrix contains the equality contraints in the first `NEQ` rows followed by the inequality constraints.

B — Vector of length `NCON` containing right-hand sides of the linear constraints. (Input)

G — Vector of length `NVAR` containing the coefficients of the linear term of the objective function. (Input)

H — `NVAR` by `NVAR` matrix containing the Hessian matrix of the objective function. (Input)
`H` should be symmetric positive definite; if `H` is not positive definite, the algorithm attempts to solve the QP problem with `H` replaced by a `H` + `DIAGNL * I` such that `H` + `DIAGNL * I` is positive definite. See Comment 3.

SOL — Vector of length `NVAR` containing solution. (Output)

Optional Arguments

NVAR — The number of variables. (Input)
Default: `NVAR` = size (`A`,2).

NCON — The number of linear constraints. (Input)
Default: `NCON` = size (`A`,1).

LDA — Leading dimension of `A` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDA` = size (`A`,1).

LDH — Leading dimension of `H` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDH` = size (`H`,1).

DIAGNL — Scalar equal to the multiple of the identity matrix added to `H` to give a positive definite matrix. (Output)

NACT — Final number of active constraints. (Output)

IACT — Vector of length `NVAR` containing the indices of the final active constraints in the first `NACT` positions. (Output)

ALAMDA — Vector of length `NVAR` containing the Lagrange multiplier estimates of the final active constraints in the first `NACT` positions. (Output)

FORTRAN 90 Interface

Generic: `CALL QPROG (NEQ, A, B, G, H, SOL [,…])`

Specific: The specific interface names are `S_QPROG` and `D_QPROG`.

FORTRAN 77 Interface

Single: `CALL QPROG (NVAR, NCON, NEQ, A, LDA, B, G, H, LDH, DIAGNL, SOL, NACT, IACT, ALAMDA)`

Double: The double precision name is `DQPROG`.

Example

The quadratic programming problem

$$\begin{aligned} &\min f(x) = x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 - 2x_2x_3 - 2x_4x_5 - 2x_1 \\ &\text{subject to} \quad x_1 + x_2 + x_3 + x_4 + x_5 = 5 \\ &\qquad\qquad\qquad x_3 - 2x_4 - 2x_5 = -3 \end{aligned}$$

is solved.

```
      USE QPROG_INT
      USE UMACH_INT
!                                  Declare variables
      INTEGER    LDA, LDH, NCON, NEQ, NVAR
      PARAMETER  (NCON=2, NEQ=2, NVAR=5, LDA=NCON, LDH=NVAR)
```

```
!
      INTEGER    K, NACT, NOUT
      REAL       A(LDA,NVAR), ALAMDA(NVAR), B(NCON), G(NVAR), &
                H(LDH,LDH), SOL(NVAR)
!
!                                 Set values of A, B, G and H.
!                                 A = ( 1.0  1.0  1.0  1.0  1.0)
!                                     ( 0.0  0.0  1.0 -2.0 -2.0)
!
!                                 B = ( 5.0 -3.0)
!
!                                 G = (-2.0  0.0  0.0  0.0  0.0)
!
!                                 H = ( 2.0  0.0  0.0  0.0  0.0)
!                                     ( 0.0  2.0 -2.0  0.0  0.0)
!                                     ( 0.0 -2.0  2.0  0.0  0.0)
!                                     ( 0.0  0.0  0.0  2.0 -2.0)
!                                     ( 0.0  0.0  0.0 -2.0  2.0)
!
      DATA A/1.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0, -2.0, 1.0, -2.0/
      DATA B/5.0, -3.0/
      DATA G/-2.0, 4*0.0/
      DATA H/2.0, 5*0.0, 2.0, -2.0, 3*0.0, -2.0, 2.0, 5*0.0, 2.0, &
          -2.0, 3*0.0, -2.0, 2.0/
!
      CALL QPROG (NEQ, A, B, G, H, SOL)
!
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) (SOL(K),K=1,NVAR)
99999 FORMAT (’  The solution vector is’, /, ’  SOL = (’, 5F6.1, &
            ’  )’)
!
      END
```

Output

```
The solution vector is
SOL = (   1.0   1.0   1.0   1.0   1.0  )
```

Comments

1. Workspace may be explicitly provided, if desired, by use of Q2ROG/DQ2ROG. The reference is:

   CALL Q2ROG (NVAR, NCON, NEQ, A, LDA, B, G, H, LDH, DIAGNL, SOL, NACT, IACT, ALAMDA, WK)

   The additional argument is:

   WK — Work vector of length (3 * NVAR**2 + 11 * NVAR)/2 + NCON.

2. Informational errors

   Type   Code

3	1	Due to the effect of computer rounding error, a change in the variables fail to improve the objective function value; usually the solution is close to optimum.
4	2	The system of equations is inconsistent. There is no solution.

3. If a perturbation of `H`, `H` + `DIAGNL * I`, was used in the `QP` problem, then `H` + `DIAGNL * I` should also be used in the definition of the Lagrange multipliers.

Description

The routine `QPROG` is based on M.J.D. Powell’s implementation of the Goldfarb and Idnani (1983) dual quadratic programming (QP) algorithm for convex QP problems subject to general linear equality/inequality constraints, i.e., problems of the form

$$\min_{x \in \mathbf{R}^n} g^T x + \frac{1}{2} x^T Hx$$

$$\text{subject to} \quad A_1 x = b_1$$

$$A_2 x \geq b_2$$

given the vectors b_1, b_2, and g and the matrices H, A_1, and A_2. H is required to be positive definite. In this case, a unique x solves the problem or the constraints are inconsistent. If H is not positive definite, a positive definite perturbation of H is used in place of H. For more details, see Powell (1983, 1985).

LCONF

Minimizes a general objective function subject to linear equality/inequality constraints.

Required Arguments

FCN — User-supplied `SUBROUTINE` to evaluate the function to be minimized. The usage is `CALL FCN (N, X, F)`, where

`N` – Value of `NVAR`.   (Input)

`X` – Vector of length `N` at which point the function is evaluated.   (Input)
`X` should not be changed by `FCN`.

`F` – The computed function value at the point `X`.   (Output)

`FCN` must be declared `EXTERNAL` in the calling program.

NEQ — The number of linear equality constraints.   (Input)

A — `NCON` by `NVAR` matrix.   (Input)
The matrix contains the equality constraint gradients in the first `NEQ` rows, followed by the inequality constraint gradients.

B — Vector of length NCON containing right-hand sides of the linear constraints. (Input)
Specifically, the constraints on the variables X(I), I = 1, …, NVAR are A(K, 1) * X(1) + … + A(K, NVAR) * X(NVAR).EQ.B(K), K = 1, …, NEQ.A(K, 1) * X(1) + … + A(K, NVAR) * X(NVAR).LE.B(K), K = NEQ + 1, …, NCON. Note that the data that define the equality constraints come before the data of the inequalities.

XLB — Vector of length NVAR containing the lower bounds on the variables; choose a very large negative value if a component should be unbounded below or set XLB(I) = XUB(I) to freeze the I-th variable. (Input)
Specifically, these simple bounds are XLB(I).LE.X(I), I = 1, …, NVAR.

XUB — Vector of length NVAR containing the upper bounds on the variables; choose a very large positive value if a component should be unbounded above. (Input)
Specifically, these simple bounds are X(I).LE.XUB(I), I = 1, …, NVAR.

SOL — Vector of length NVAR containing solution. (Output)

Optional Arguments

NVAR — The number of variables. (Input)
Default: NVAR = size (A,2).

NCON — The number of linear constraints (excluding simple bounds). (Input)
Default: NCON = size (A,1).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

XGUESS — Vector of length NVAR containing the initial guess of the minimum. (Input)
Default: XGUESS = 0.0.

ACC — The nonnegative tolerance on the first order conditions at the calculated solution. (Input)
Default: ACC = 1.e-4 for single precision and 1.d-8 for double precision.

MAXFCN — On input, maximum number of function evaluations allowed. (Input/ Output)
On output, actual number of function evaluations needed.
Default: MAXFCN = 400.

OBJ — Value of the objective function. (Output)

NACT — Final number of active constraints. (Output)

IACT — Vector containing the indices of the final active constraints in the first NACT positions. (Output)
Its length must be at least NCON + 2 * NVAR.

ALAMDA — Vector of length `NVAR` containing the Lagrange multiplier estimates of the final active constraints in the first `NACT` positions. (Output)

FORTRAN 90 Interface

Generic: `CALL LCONF (FCN, NEQ, A, B, XLB, XUB, SOL [,…])`

Specific: The specific interface names are `S_LCONF` and `D_LCONF`.

FORTRAN 77 Interface

Single:

```
CALL LCONF (FCN, NVAR, NCON, NEQ, A, LDA, B, XLB, XUB,
XGUESS, ACC, MAXFCN, SOL, OBJ, NACT, IACT,
ALAMDA)
```

Double: The double precision name is `DLCONF`.

Example

The problem from Schittkowski (1987)

$$\min f(x) = -x_1 x_2 x_3$$

$$\begin{aligned} \text{subject to} \quad & -x_1 - 2x_2 - 2x_3 \le 0 \\ & x_1 + 2x_2 + 2x_3 \le 72 \\ & 0 \le x_1 \le 20 \\ & 0 \le x_2 \le 11 \\ & 0 \le x_3 \le 42 \end{aligned}$$

is solved with an initial guess $x_1 = 10$, $x_2 = 10$ and $x_3 = 10$.

```
      USE LCONF_INT
      USE UMACH_INT
!                                 Declaration of variables
      INTEGER    NCON, NEQ, NVAR
      PARAMETER  (NCON=2, NEQ=0, NVAR=3)
!
      INTEGER    MAXFCN, NOUT
      REAL       A(NCON,NVAR), ACC, B(NCON), OBJ, &
                SOL(NVAR), XGUESS(NVAR), XLB(NVAR), XUB(NVAR)
      EXTERNAL   FCN
!
!                                 Set values for the following problem.
!
!                                 Min  -X(1)*X(2)*X(3)
!
!                                 -X(1) - 2*X(2) - 2*X(3)  .LE.   0
!                                  X(1) + 2*X(2) + 2*X(3)  .LE.  72
!
```

```
!                                 0  .LE.  X(1)  .LE.  20
!                                 0  .LE.  X(2)  .LE.  11
!                                 0  .LE.  X(3)  .LE.  42
!
      DATA A/-1.0, 1.0, -2.0, 2.0, -2.0, 2.0/, B/0.0, 72.0/
      DATA XLB/3*0.0/, XUB/20.0, 11.0, 42.0/, XGUESS/3*10.0/
      DATA ACC/0.0/, MAXFCN/400/
!
      CALL UMACH (2, NOUT)
!
      CALL LCONF (FCN, NEQ, A, B, XLB, XUB, SOL, XGUESS=XGUESS,  &
                 MAXFCN=MAXFCN, ACC=ACC, OBJ=OBJ)
!
      WRITE (NOUT,99998) 'Solution:'
      WRITE (NOUT,99999) SOL
      WRITE (NOUT,99998) 'Function value at solution:'
      WRITE (NOUT,99999) OBJ
      WRITE (NOUT,99998) 'Number of function evaluations:', MAXFCN
      STOP
99998 FORMAT (//, ' ', A, I4)
99999 FORMAT (1X, 5F16.6)
      END
!
      SUBROUTINE FCN (N, X, F)
      INTEGER    N
      REAL       X(*), F
!
      F = -X(1)*X(2)*X(3)
      RETURN
      END
```

Output

```
Solution:
 20.000000       11.000000       15.000000

Function value at solution:
-3300.000000

Number of function evaluations:   5
```

Comments

1. Workspace may be explicitly provided, if desired, by use of L2ONF/DL2ONF. The reference is:

   ```
   CALL L2ONF (FCN, NVAR, NCON, NEQ, A, LDA, B, XLB, XUB, XGUESS,
   ACC, MAXFCN, SOL, OBJ, NACT, IACT, ALAMDA, IPRINT, INFO, WK)
   ```

   The additional arguments are as follows:

   IPRINT — Print option (see Comment 3). (Input)

   INFO — Informational flag (see Comment 3). (Output)

WK — Real work vector of length NVAR**2 + 11 * NVAR + NCON.

2. Informational errors

Type	Code	
4	4	The equality constraints are inconsistent.
4	5	The equality constraints and the bounds on the variables are found to be inconsistent.
4	6	No vector X satisfies all of the constraints. In particular, the current active constraints prevent any change in X that reduces the sum of constraint violations.
4	7	Maximum number of function evaluations exceeded.
4	9	The variables are determined by the equality constraints.

3. The following are descriptions of the arguments IPRINT and INFO:

IPRINT — This argument must be set by the user to specify the frequency of printing during the execution of the routine LCONF. There is no printed output if IPRINT = 0. Otherwise, after ensuring feasibility, information is given every IABS(IPRINT) iterations and whenever a parameter called TOL is reduced. The printing provides the values of X(.), F(.) and G(.) = GRAD(F) if IPRINT is positive. If IPRINT is negative, this information is augmented by the current values of IACT(K) K = 1, …, NACT, PAR(K) K = 1, …, NACT and RESKT(I) I = 1, …, N. The reason for returning to the calling program is also displayed when IPRINT is nonzero.

INFO — On exit from L2ONF, INFO will have one of the following integer values to indicate the reason for leaving the routine:

INFO = 1 SOL is feasible, and the condition that depends on ACC is satisfied.

INFO = 2 SOL is feasible, and rounding errors are preventing further progress.

INFO = 3 SOL is feasible, but the objective function fails to decrease although a decrease is predicted by the current gradient vector.

INFO = 4 In this case, the calculation cannot begin because LDA is less than NCON or because the lower bound on a variable is greater than the upper bound.

INFO = 5 This value indicates that the equality constraints are inconsistent. These constraints include any components of X(.) that are frozen by setting XL(I) = XU(I).

INFO = 6 In this case there is an error return because the equality constraints and the bounds on the variables are found to be inconsistent.

INFO = 7 This value indicates that there is no vector of variables that satisfies all of the constraints. Specifically, when this return or an INFO = 6 return occurs, the current active constraints (whose indices are IACT(K), K = 1, …, NACT) prevent any change in X(.) that reduces the sum of constraint violations. Bounds are only included in this sum if INFO = 6.

INFO = 8 Maximum number of function evaluations exceeded.

INFO = 9 The variables are determined by the equality constraints.

Description

The routine LCONF is based on M.J.D. Powell's TOLMIN, which solves linearly constrained optimization problems, i.e., problems of the form

$$\min_{x \in \mathbf{R}^n} f(x)$$

$$\text{subject to} \quad A_1 x = b_1$$

$$A_2 x \le b_2$$

$$x_l \le x \le x_u$$

given the vectors b_1, b_2, x_l and x_u and the matrices A_1, and A_2.

The algorithm starts by checking the equality constraints for inconsistency and redundancy. If the equality constraints are consistent, the method will revise x^0, the initial guess provided by the user, to satisfy

$$A_1 x = b_1$$

Next, x^0 is adjusted to satisfy the simple bounds and inequality constraints. This is done by solving a sequence of quadratic programming subproblems to minimize the sum of the constraint or bound violations.

Now, for each iteration with a feasible x^k, let J_k be the set of indices of inequality constraints that have small residuals. Here, the simple bounds are treated as inequality constraints. Let I_k be the set of indices of active constraints. The following quadratic programming problem

$$\min f\left(x^k\right) + d^T \nabla f\left(x^k\right) + \frac{1}{2} d^T B^k d$$

$$\text{subject to} \quad a_j d = 0 \;\; j \in I_k$$

$$a_j d \le 0 \;\; j \in J_k$$

is solved to get (d^k, λ^k) where a_j is a row vector representing either a constraint in A_1 or A_2 or a bound constraint on x. In the latter case, the $a_j = e_i$ for the bound constraint $x_i \le (x_u)_i$ and $a_j = -e_i$ for the constraint $-x_i \le (-x_l)_i$. Here, e_i is a vector with a 1 as the i-th component, and zeroes elsewhere. λ^k are the Lagrange multipliers, and B^k is a positive definite approximation to the second derivative $\nabla^2 f(x^k)$.

After the search direction d^k is obtained, a line search is performed to locate a better point. The new point $x^{k+1} = x^k + \alpha^k d^k$ has to satisfy the conditions

$$f\left(x^k + \alpha^k d^k\right) \le f\left(x^k\right) + 0.1\alpha^k \left(d^k\right)^T \nabla f\left(x^k\right)$$

and

$$\left(d^k\right)^T \nabla f\left(x^k + \alpha^k d^k\right) \ge 0.7\left(d^k\right)^T \nabla f\left(x^k\right)$$

The main idea in forming the set J_k is that, if any of the inequality constraints restricts the step-length α^k, then its index is not in J_k. Therefore, small steps are likely to be avoided.

Finally, the second derivative approximation, B^k , is updated by the BFGS formula, if the condition

$$\left(d^k\right)^T \nabla f\left(x^k + \alpha^k d^k\right) - \nabla f\left(x^k\right) > 0$$

holds. Let $x^k \leftarrow x^{k+1}$, and start another iteration.

The iteration repeats until the stopping criterion

$$\left\| \nabla f\left(x^k\right) - A^k \lambda^k \right\|_2 \le \tau$$

is satisfied; here, τ is a user-supplied tolerance. For more details, see Powell (1988, 1989).

Since a finite-difference method is used to estimate the gradient for some single precision calculations, an inaccurate estimate of the gradient may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended. Also, whenever the exact gradient can be easily provided, routine LCONG (page 1316) should be used instead.

LCONG

Minimizes a general objective function subject to linear equality/inequality constraints.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function to be minimized. The usage is CALL FCN (N, X, F), where

N – Value of NVAR. (Input)

X – Vector of length N at which point the function is evaluated. (Input)
X should not be changed by FCN.

F – The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

GRAD — User-supplied SUBROUTINE to compute the gradient at the point X. The usage is CALL GRAD (N, X, G), where

N – Value of NVAR. (Input)

X – Vector of length N at which point the function is evaluated. (Input)
X should not be changed by GRAD.

G – Vector of length N containing the values of the gradient of the objective function evaluated at the point X. (Output)

GRAD must be declared EXTERNAL in the calling program.

NEQ — The number of linear equality constraints. (Input)

A — NCON by NVAR matrix. (Input)
The matrix contains the equality constraint gradients in the first NEQ rows, followed by the inequality constraint gradients.

B — Vector of length NCON containing right-hand sides of the linear constraints. (Input)
Specifically, the constraints on the variables X(I), I = 1, …, NVAR are A(K, 1) * X(1) + … + A(K, NVAR) * X(NVAR).EQ.B(K), K = 1, …, NEQ.A(K, 1) * X(1) + … + A(K, NVAR) * X(NVAR).LE.B(K), K = NEQ + 1, …, NCON. Note that the data that define the equality constraints come before the data of the inequalities.

XLB — Vector of length NVAR containing the lower bounds on the variables; choose a very large negative value if a component should be unbounded below or set XLB(I) = XUB(I) to freeze the I-th variable. (Input)
Specifically, these simple bounds are XLB(I).LE.X(I), I = 1, …, NVAR.

XUB — Vector of length NVAR containing the upper bounds on the variables; choose a very large positive value if a component should be unbounded above. (Input)
Specifically, these simple bounds are X(I).LE. XUB(I), I = 1, …, NVAR.

SOL — Vector of length NVAR containing solution. (Output)

Optional Arguments

NVAR — The number of variables. (Input)
Default: NVAR = size (A,2).

NCON — The number of linear constraints (excluding simple bounds). (Input)
Default: NCON = size (A,1).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

XGUESS — Vector of length NVAR containing the initial guess of the minimum. (Input)
Default: XGUESS = 0.0.

ACC — The nonnegative tolerance on the first order conditions at the calculated solution. (Input)
Default: ACC = 1.e-4 for single precision and 1.d-8 for double precision.

MAXFCN — On input, maximum number of function evaluations allowed.(Input/ Output)
On output, actual number of function evaluations needed.
Default: MAXFCN = 400.

OBJ — Value of the objective function. (Output)

NACT — Final number of active constraints. (Output)

IACT — Vector containing the indices of the final active constraints in the first NACT positions. (Output)
Its length must be at least NCON + 2 * NVAR.

ALAMDA — Vector of length NVAR containing the Lagrange multiplier estimates of the final active constraints in the first NACT positions. (Output)

FORTRAN 90 Interface

Generic: `CALL LCONG (FCN, GRAD, NEQ, A, B, XLB, XUB, SOL [,…])`

Specific: The specific interface names are S_LCONG and D_LCONG.

FORTRAN 77 Interface

Single:
```
CALL LCONG (FCN, GRAD, NVAR, NCON, NEQ, A, LDA, B, XLB,
XUB, XGUESS, ACC, MAXFCN, SOL, OBJ, NACT, IACT,
ALAMDA)
```

Double: The double precision name is DLCONG.

Example

The problem from Schittkowski (1987)

$$\min f(x) = -x_1 x_2 x_3$$

$$\text{subject to} \quad -x_1 - 2x_2 - 2x_3 \le 0$$

$$x_1 + 2x_2 + 2x_3 \le 72$$

$$0 \le x_1 \le 20$$

$$0 \le x_2 \le 11$$

$$0 \le x_3 \le 42$$

is solved with an initial guess $x_1 = 10$, $x_2 = 10$ and $x_3 = 10$.

```
      USE LCONG_INT
      USE UMACH_INT
!                                 Declaration of variables
      INTEGER    NCON, NEQ, NVAR
      PARAMETER  (NCON=2, NEQ=0, NVAR=3)
!
      INTEGER    MAXFCN, NOUT
      REAL       A(NCON,NVAR), ACC, B(NCON), OBJ, &
                 SOL(NVAR), XGUESS(NVAR), XLB(NVAR), XUB(NVAR)
      EXTERNAL   FCN, GRAD
!
!                                 Set values for the following problem.
!
!                                 Min  -X(1)*X(2)*X(3)
!
!                                 -X(1) - 2*X(2) - 2*X(3)  .LE.   0
!                                  X(1) + 2*X(2) + 2*X(3)  .LE.  72
!
!                                 0  .LE.  X(1)  .LE.  20
!                                 0  .LE.  X(2)  .LE.  11
!                                 0  .LE.  X(3)  .LE.  42
!
      DATA A/-1.0, 1.0, -2.0, 2.0, -2.0, 2.0/, B/0.0, 72.0/
      DATA XLB/3*0.0/, XUB/20.0, 11.0, 42.0/, XGUESS/3*10.0/
      DATA ACC/0.0/, MAXFCN/400/
!
      CALL UMACH (2, NOUT)
!
      CALL LCONG (FCN, GRAD, NEQ, A, B, XLB, XUB, SOL, XGUESS=XGUESS, &
                  ACC=ACC, MAXFCN=MAXFCN, OBJ=OBJ)
!
      WRITE (NOUT,99998) 'Solution:'
      WRITE (NOUT,99999) SOL
      WRITE (NOUT,99998) 'Function value at solution:'
      WRITE (NOUT,99999) OBJ
      WRITE (NOUT,99998) 'Number of function evaluations:', MAXFCN
      STOP
99998 FORMAT (//, ' ', A, I4)
99999 FORMAT (1X, 5F16.6)
      END
!
      SUBROUTINE FCN (N, X, F)
      INTEGER    N
      REAL       X(*), F
!
      F = -X(1)*X(2)*X(3)
      RETURN
      END
!
      SUBROUTINE GRAD (N, X, G)
      INTEGER    N
      REAL       X(*), G(*)
```

```
!
      G(1) = -X(2)*X(3)
      G(2) = -X(1)*X(3)
      G(3) = -X(1)*X(2)
      RETURN
      END
```

Output

```
Solution:
20.000000       11.000000       15.000000

Function value at solution:
-3300.000000

Number of function evaluations:   5
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `L2ONG/DL2ONG`. The reference is:

   ```
   CALL L2ONG (FCN, GRAD, NVAR, NCON, NEQ, A, LDA, B, XLB, XUB,
   XGUESS, ACC, MAXFCN, SOL, OBJ, NACT, IACT, ALAMDA, IPRINT,
   INFO, WK)
   ```

   The additional arguments are as follows:

   IPRINT — Print option (see Comment 3). (Input)

   INFO — Informational flag (see Comment 3). (Output)

   WK — Real work vector of length `NVAR**2 + 11 * NVAR + NCON.`

2. Informational errors

Type	Code	
4	4	The equality constraints are inconsistent.
4	5	The equality constraints and the bounds on the variables are found to be inconsistent.
4	6	No vector `X` satisfies all of the constraints. In particular, the current active constraints prevent any change in `X` that reduces the sum of constraint violations.
4	7	Maximum number of function evaluations exceeded.
4	9	The variables are determined by the equality constraints.

3. The following are descriptions of the arguments `IPRINT` and `INFO`:

   IPRINT — This argument must be set by the user to specify the frequency of printing during the execution of the routine `LCONG`. There is no printed output if `IPRINT` = 0. Otherwise, after ensuring feasibility, information is given every `IABS`(`IPRINT`) iterations and whenever a parameter called TOL is reduced. The printing provides the values of `X`(.), `F`(.) and `G`(.) = `GRAD`(F) if `IPRINT` is positive. If IPRINT is negative, this information is augmented by the current values of IACT(K) K = 1, …, NACT, PAR(K) K = 1, …, NACT and RESKT(I) I = 1, …, N. The reason for returning to the calling program is also displayed when IPRINT is nonzero.

INFO — On exit from L2ONG, INFO will have one of the following integer values to indicate the reason for leaving the routine:

INFO = 1 SOL is feasible and the condition that depends on ACC is satisfied.

INFO = 2 SOL is feasible and rounding errors are preventing further progress.

INFO = 3 SOL is feasible but the objective function fails to decrease although a decrease is predicted by the current gradient vector.

INFO = 4 In this case, the calculation cannot begin because LDA is less than NCON or because the lower bound on a variable is greater than the upper bound.

INFO = 5 This value indicates that the equality constraints are inconsistent. These constraints include any components of X(.) that are frozen by setting XL(I) = XU(I).

INFO = 6 In this case, there is an error return because the equality constraints and the bounds on the variables are found to be inconsistent.

INFO = 7 This value indicates that there is no vector of variables that satisfies all of the constraints. Specifically, when this return or an INFO = 6 return occurs, the current active constraints (whose indices are IACT(K), K = 1, …, NACT) prevent any change in X(.) that reduces the sum of constraint violations, where only bounds are included in this sum if INFO = 6.

INFO = 8 Maximum number of function evaluations exceeded.

INFO = 9 The variables are determined by the equality constraints.

Description

The routine LCONG is based on M.J.D. Powell’s TOLMIN, which solves linearly constrained optimization problems, i.e., problems of the form

$$\min_{x \in \mathbf{R}^n} f(x)$$

$$\text{subject to} \quad A_1 x = b_1$$

$$A_2 x \le b_2$$

$$x_l \le x \le x_u$$

given the vectors b_1, b_2, x_l and x_u and the matrices A_1, and A_2.

The algorithm starts by checking the equality constraints for inconsistency and redundancy. If the equality constraints are consistent, the method will revise x^0, the initial guess provided by the user, to satisfy

$$A_1 x = b_1$$

Next, x^0 is adjusted to satisfy the simple bounds and inequality constraints. This is done by solving a sequence of quadratic programming subproblems to minimize the sum of the constraint or bound violations.

Now, for each iteration with a feasible x_k, let J_k be the set of indices of inequality constraints that have small residuals. Here, the simple bounds are treated as inequality constraints. Let I_k be the set of indices of active constraints. The following quadratic programming problem

$$\min f\left(x^k\right) + d^T \nabla f\left(x^k\right) + \frac{1}{2} d^T B^k d$$

$$\text{subject to} \quad a_j d = 0 \quad j \in I_k$$

$$a_j d \le 0 \quad j \in J_k$$

is solved to get (d^k, λ^k) where a_j is a row vector representing either a constraint in A_1 or A_2 or a bound constraint on x. In the latter case, the $a_j = e_i$ for the bound constraint $x_i \le (x_u)_i$ and $a_j = -e_i$ for the constraint $-x_i \le (-x_l)_i$. Here, e_i is a vector with a 1 as the i-th component, and zeroes elsewhere. λ^k are the Lagrange multipliers, and B^k is a positive definite approximation to the second derivative $\nabla^2 f(x^k)$.

After the search direction d^k is obtained, a line search is performed to locate a better point. The new point $x^{k+1} = x^k + \alpha^k d^k$ has to satisfy the conditions

$$f\left(x^k + \alpha^k d^k\right) \le f\left(x^k\right) + 0.1\alpha^k \left(d^k\right)^T \nabla f\left(x^k\right)$$

and

$$\left(d^k\right)^T \nabla f\left(x^k + \alpha^k d^k\right) \ge 0.7\left(d^k\right)^T \nabla f\left(x^k\right)$$

The main idea in forming the set J_k is that, if any of the inequality constraints restricts the step-length α^k, then its index is not in J_k. Therefore, small steps are likely to be avoided.

Finally, the second derivative approximation, B^k, is updated by the BFGS formula, if the condition

$$\left(d^k\right)^T \nabla f\left(x^k + \alpha^k d^k\right) - \nabla f\left(x^k\right) > 0$$

holds. Let $x^k \leftarrow x^{k+1}$, and start another iteration.

The iteration repeats until the stopping criterion

$$\left\|\nabla f\left(x^k\right) - A^k \lambda^k\right\|_2 \le \tau$$

is satisfied; here, τ is a user-supplied tolerance. For more details, see Powell (1988, 1989).

NNLPF

Solves a general nonlinear programming problem using a sequential equality constrained quadratic programming method.

Required Arguments

FCN — User-supplied `SUBROUTINE` to evaluate the objective function and constraints at a given point. The internal usage is `CALL FCN (X, IACT, RESULT, IERR)`, where

X – The point at which the objective function or constraint is evaluated.   (Input)

IACT – Integer indicating whether evaluation of the objective function is requested or evaluation of a constraint is requested. If `IACT` is zero, then an objective function evaluation is requested. If `IACT` is nonzero then the value if `IACT` indicates the index of the constraint to evaluate.   (Input)

RESULT – If `IACT` is zero, then `RESULT` is the computed function value at the point `X`. If `IACT` is nonzero, then `RESULT` is the computed constraint value at the point `X`.   (Output)

IERR – Logical variable. On input `IERR` is set to `.FALSE.` If an error or other undesirable condition occurs during evaluation, then `IERR` should be set to `.TRUE.` Setting `IERR` to `.TRUE.` will result in the step size being reduced and the step being tried again. (If `IERR` is set to `.TRUE.` for `XGUESS`, then an error is issued.)

The routine `FCN` must be use-associated in a user module that uses `NNLPF_INT`, or else declared `EXTERNAL` in the calling program. If `FCN` is a separately compiled routine, not in a module, then it must be declared `EXTERNAL`.

M — Total number of constraints. (Input)

ME — Number of equality constraints. (Input)

IBTYPE — Scalar indicating the types of bounds on variables.   (Input)

`IBTYPE`	**Action**
0	User will supply all the bounds.

1	All variables are nonnegative.
2	All variables are nonpositive.
3	User supplies only the bounds on 1st variable; all other variables will have the same bounds.

XLB — Vector of length `N` containing the lower bounds on variables. (Input, if `IBTYPE` = 0; output, if `IBTYPE` = 1 or 2; input/output, if `IBTYPE` = 3)
If there is no lower bound for a variable, then the corresponding `XLB` value should be set to −`Huge`(`X`(`1`)).

XUB — Vector of length `N` containing the upper bounds on variables. (Input, if `IBTYPE` = 0; output, if `IBTYPE` = 1 or 2; input/output, if `IBTYPE` = 3).
If there is no upper bound for a variable, then the corresponding `XUB` value should be set to `Huge`(`X`(`1`)).

X — Vector of length `N` containing the computed solution. (Output)

Optional Arguments

N — Number of variables. (Input)
Default: `N` = `size`(`X`).

XGUESS — Vector of length `N` containing an initial guess of the solution. (Input)
Default: `XGUESS` = `X`, (with the smallest value of $\|X\|_2$) that satisfies the bounds.

XSCALE — Vector of length `N` setting the internal scaling of the variables. The initial value given and the objective function and gradient evaluations however are always in the original unscaled variables. The first internal variable is obtained by dividing values `X(I)` by `XSCALE(I)`. (Input)
In the absence of other information, set all entries to 1.0.
Default: `XSCALE(:)` = 1.0.

IPRINT — Parameter indicating the desired output level. (Input)

`IPRINT`	**Action**
0	No output printed.
1	One line of intermediate results is printed in each iteration.
2	Lines of intermediate results summarizing the most important data for each step are printed.

3 Lines of detailed intermediate results showing all primal and dual variables, the relevant values from the working set, progress in the backtracking and etc are printed

4 Lines of detailed intermediate results showing all primal and dual variables, the relevant values from the working set, progress in the backtracking, the gradients in the working set, the quasi-Newton updated and etc are printed.

Default: `IPRINT` = 0.

MAXITN — Maximum number of iterations allowed. (Input)
Default: `MAXITN` = 200.

EPSDIF — Relative precision in gradients. (Input)
Default: `EPSDIF` = `epsilon`(`x`(1))

TAU0 — A universal bound describing how much the unscaled penalty-term may deviate from zero. (Input)
`NNLPF` assumes that within the region described by

$$\sum_{i=1}^{M_e} \left|g_i\left(x\right)\right| - \sum_{i=M_e+1}^{M} \min\left(0, g_i\left(x\right)\right) \le \texttt{TAU0}$$

all functions may be evaluated safely. The initial guess, however, may violate these requirements. In that case an initial feasibility improvement phase is run by `NNLPF` until such a point is found. A small `TAU0` diminishes the efficiency of `NNLPF`, because the iterates then will follow the boundary of the feasible set closely. Conversely, a large `TAU0` may degrade the reliability of the code.
Default `TAU0` = 1.E0

DEL0 — In the initial phase of minimization a constraint is considered binding if

$$\frac{g_i\left(x\right)}{\max\left(1, \left\|\nabla g_i\left(x\right)\right\|\right)} \le \texttt{DEL0} \qquad i = M_e + 1, \ldots, M$$

Good values are between .01 and 1.0. If `DEL0` is chosen too small then identification of the correct set of binding constraints may be delayed. Contrary, if `DEL0` is too large, then the method will often escape to the full regularized SQP method, using individual slack variables for any active constraint, which is quite costly. For well-scaled problems `DEL0=1.0` is reasonable. (Input)
Default: `DEL0` = .5*`TAU0`

EPSFCN **–** Relative precision of the function evaluation routine. (Input)
Default: `EPSFCN` = `epsilon`(`x`(1))

IDTYPE **–** Type of numerical differentiation to be used. (Input)
Default: `IDTYPE` = 1

IDTYPE	**Action**
1	Use a forward difference quotient with discretization stepsize 0.1 (EPSFCN$^{1/2}$) componentwise relative.
2	Use the symmetric difference quotient with discretization stepsize 0.1 (EPSFCN$^{1/3}$) componentwise relative
3	Use the sixth order approximation computing a Richardson extrapolation of three symmetric difference quotient values. This uses a discretization stepsize 0.01(EPSFCN$^{1/7}$)

TAUBND **–** Amount by which bounds may be violated during numerical differentiation. Bounds are violated by TAUBND (at most) only if a variable is on a bound and finite differences are taken for gradient evaluations. (Input)
Default: TAUBND = 1.E0

SMALLW — Scalar containing the error allowed in the multipliers. For example, a negative multiplier of an inequality constraint is accepted (as zero) if its absolute value is less than SMALLW. (Input)
Default: SMALLW = exp(2*log(epsilon(x(1)/3)))

DELMIN — Scalar which defines allowable constraint violations of the final accepted result. Constraints are satisfied if $|g_i(x)| \leq$ DELMIN , and $g_j(x) \geq$ (-DELMIN) respectively. (Input)
Default: DELMIN = min(DEL0/10, max(EPSDIF, min(DEL0/10, max(1.E-6*DEL0, SMALLW))

SCFMAX — Scalar containing the bound for the internal automatic scaling of the objective function. (Intput)
Default: SCFMAX = 1.0E4

FVALUE — Scalar containing the value of the objective function at the computed solution. (Output)

FORTRAN 90 Interface

Generic: CALL NNLPF (FCN, M, ME, IBTYPE, XLB, XUB, X [,…])

Specific: The specific interface names are S_NNLPF and D_NNLPF .

Example

The problem

$$\min F(x) = (x_1 - 2)^2 + (x_2 - 1)^2$$

$$\text{subject to} \quad g_1(x) = x_1 - 2x_2 + 1 = 0$$

$$g_2(x) = -x_1^2 / 4 - x_2^2 + 1 \ge 0$$

is solved.

```
      USE NNLPF_INT
      USE WRRRN_INT
      INTEGER    IBTYPE, M, ME
      PARAMETER  (IBTYPE=0, M=2, ME=1)
!
      REAL(KIND(1E0)) FVALUE, X(2), XGUESS(2), XLB(2), XUB(2)
      EXTERNAL FCN, GRAD
!
      XLB = -HUGE(X(1))
      XUB = HUGE(X(1))
!
      CALL NNLPF (FCN, M, ME, IBTYPE, XLB, XUB, X)
!
      CALL WRRRN ('The solution is', X)
      END

      SUBROUTINE FCN (X, IACT, RESULT, IERR)
      INTEGER    IACT
      REAL(KIND(1E0)) X(*), RESULT
      LOGICAL IERR
!
      SELECT CASE (IACT)
      CASE(0)
         RESULT = (X(1)-2.0E0)**2 + (X(2)-1.0E0)**2
      CASE(1)
         RESULT = X(1) - 2.0E0*X(2) + 1.0E0
      CASE(2)
         RESULT = -(X(1)**2)/4.0E0 - X(2)**2 + 1.0E0
      END SELECT
      RETURN
      END
```

Output

```
The solution is
 1   0.8229
 2   0.9114
```

Comments

1. Informational errors

Type	Code	
4	1	Constraint evaluation returns an error with current point.
4	2	Objective evaluation returns an error with current point.
4	3	Working set is singular in dual extended QP.
4	4	QP problem is seemingly infeasible.
4	5	A stationary point located.

4	6	A stationary point located or termination criteria too strong.
4	7	Maximum number of iterations exceeded.
4	8	Stationary point not feasible.
4	9	Very slow primal progress.
4	10	The problem is singular.
4	11	Matrix of gradients of binding constraints is singular or very ill-conditioned.
4	12	Small changes in the penalty function.

Description

The routine `NNLPF` provides an interface to a licensed version of subroutine `DONLP2`, a FORTRAN code developed by Peter Spellucci (1998). It uses a sequential equality constrained quadratic programming method with an active set technique, and an alternative usage of a fully regularized mixed constrained subproblem in case of nonregular constraints (i.e. linear dependent gradients in the "working sets"). It uses a slightly modified version of the Pantoja-Mayne update for the Hessian of the Lagrangian, variable dual scaling and an improved Armjijo-type stepsize algorithm. Bounds on the variables are treated in a gradient-projection like fashion. Details may be found in the following two papers:

P. Spellucci: *An SQP method for general nonlinear programs using only equality constrained subproblems*. Math. Prog. 82, (1998), 413-448.

P. Spellucci: *A new technique for inconsistent problems in the SQP method.* Math. Meth. of Oper. Res. 47, (1998), 355-500. (published by Physica Verlag, Heidelberg, Germany).

The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

$$\text{subject to} \quad \begin{aligned} &g_j(x) = 0, \text{ for} && j = 1, \ldots, m_e \\ &g_j(x) \ge 0, \text{ for} && j = m_e + 1, \ldots, m \\ &x_l \le x \le x_u \end{aligned}$$

Although default values are provided for optional input arguments, it may be necessary to adjust these values for some problems. Through the use of optional arguments, NNLPF allows for several parameters of the algorithm to be adjusted to account for specific characteristics of problems. The DONLP2 Users Guide provides detailed descriptions of these parameters as well as strategies for maximizing the perfomance of the algorithm. The DONLP2 Users Guide is available in the "*help*" subdirectory of the main IMSL product installation directory. In addition, the following are a number of guidelines to consider when using NNLPF.

- A good initial starting point is very problem specific and should be provided by the calling program whenever possible. See optional argument `XGUESS`.
- Gradient approximation methods can have an effect on the success of NNLPF. Selecting a higher order appoximation method may be necessary for some problems. See optional argument `IDTYPE`.

- If a two sided constraint $l_i \le g_i(x) \le u_i$ is transformed into two constraints $g_{2i}(x) \ge 0$ and $g_{2i+1}(x) \ge 0$, then choose $\mathrm{DEL0} < \frac{1}{2}(u_i - l_i) / max\{1, \|\nabla g_i(x)\|\}$, or at least try to provide an estimate for that value. This will increase the efficiency of the algorithm. See optional argument `DEL0`.
- The parameter `IERR` provided in the interface to the user supplied function `FCN` can be very useful in cases when evaluation is requested at a point that is not possible or reasonable. For example, if evaluation at the requested point would result in a floating point exception, then setting `IERR` to `.TRUE.` and returning without performing the evaluation will avoid the exception. NNLPF will then reduce the stepsize and try the step again. Note, if `IERR` is set to `.TRUE.` for the initial guess, then an error is issued.

NNLPG

Solves a general nonlinear programming problem using a sequential equality constrained quadratic programming method with user supplied gradients.

Required Arguments

FCN — User-supplied `SUBROUTINE` to evaluate the objective function and constraints at a given point. The internal usage is `CALL FCN (X, IACT, RESULT, IERR)`, where

X – The point at which the objective function or constraint is evaluated. (Input)

IACT – Integer indicating whether evaluation of the objective function is requested or evaluation of a constraint is requested. If `IACT` is zero, then an objective function evaluation is requested. If `IACT` is nonzero then the value if `IACT` indicates the index of the constraint to evaluate. (Input)

RESULT – If `IACT` is zero, then `RESULT` is the computed objective function value at the point `X`. If `IACT` is nonzero, then `RESULT` is the computed constraint value at the point `X`. (Output)

IERR – Logical variable. On input `IERR` is set to `.FALSE.` If an error or other undesirable condition occurs during evaluation, then `IERR` should be set to `.TRUE.` Setting `IERR` to `.TRUE.` will result in the step size being reduced and the step being tried again. (If `IERR` is set to `.TRUE.` for `XGUESS`, then an error is issued.)

The routine `FCN` must be use-associated in a user module that uses `NNLPG_INT`, or else declared `EXTERNAL` in the calling program. If `FCN` is a separately compiled routine, not in a module, then it must be declared `EXTERNAL`.

GRAD — User-supplied `SUBROUTINE` to evaluate the gradients at a given point. The usage is `CALL GRAD (X, IACT, RESULT)`, where

X – The point at which the gradient of the objective function or gradient of a constraint is evaluated.   (Input)

IACT – Integer indicating whether evaluation of the function gradient is requested or evaluation of a constraint gradient is requested. If `IACT` is zero, then an objective function gradient evaluation is requested. If `IACT` is nonzero then the value if `IACT` indicates the index of the constraint gradient to evaluate. (Input)***RESULT*** – If `IACT` is zero, then `RESULT` is the computed gradient of the objective function at the point `X`.   If `IACT` is nonzero, then `RESULT` is the computed gradient of the requested constraint value at the point `X`.   (Output)

The routine `GRAD` must be use-associated in a user module that uses `NNLPG_INT`, or else declared `EXTERNAL` in the calling program. If `GRAD` is a separately compiled routine, not in a module, then is must be declared `EXTERNAL`

M — Total number of constraints. (Input)

ME — Number of equality constraints. (Input)

IBTYPE — Scalar indicating the types of bounds on variables.   (Input)

`IBTYPE`	**Action**
0	User will supply all the bounds.
1	All variables are nonnegative.
2	All variables are nonpositive.
3	User supplies only the bounds on 1st variable, all other variables will have the same bounds.

XLB — Vector of length `N` containing the lower bounds on the variables.   (Input, if `IBTYPE` = 0; output, if `IBTYPE` = 1 or 2; input/output, if `IBTYPE` = 3) If there is no lower bound on a variable, then the corresponding `XLB` value should be set to `−huge(x(1))`.

XUB — Vector of length `N` containing the upper bounds on the variables.   (Input, if `IBTYPE` = 0; output, if `IBTYPE` = 1 or 2; input/output, if `IBTYPE` = 3) If there is no upper bound on a variable, then the corresponding `XUB` value should be set to `huge(x(1))`.

X — Vector of length `N` containing the computed solution.   (Output)

Optional Arguments

N — Number of variables.   (Input)
Default: `N` = `size(X)`.

IPRINT — Parameter indicating the desired output level. (Input)

`IPRINT`	**Action**
0	No output printed.
1	One line of intermediate results is printed in each iteration.
2	Lines of intermediate results summarizing the most important data for each step are printed.
3	Lines of detailed intermediate results showing all primal and dual variables, the relevant values from the working set, progress in the backtracking and etc are printed
4	Lines of detailed intermediate results showing all primal and dual variables, the relevant values from the working set, progress in the backtracking, the gradients in the working set, the quasi-Newton updated and etc are printed.

Default: `IPRINT` = 0.

MAXITN — Maximum number of iterations allowed. (Input)
Default: `MAXITN` = 200.

XGUESS — Vector of length `N` containing an initial guess of the solution. (Input)
Default: `XGUESS` = `X`, (with the smallest value of $\|X\|_2$) that satisfies the bounds.

TAU0 — A universal bound describing how much the unscaled penalty-term may deviate from zero. (Input)
`NNLPG` assumes that within the region described by

$$\sum_{i=1}^{M_e} \left|g_i(x)\right| - \sum_{i=M_e+1}^{M} \min\left(0, g_i(x)\right) \le \mathtt{TAU0}$$

all functions may be evaluated safely. The initial guess however, may violate these requirements. In that case an initial feasibility improvement phase is run by `NNLPG` until such a point is found. A small `TAU`0 diminishes the efficiency of `NNLPG`, because the iterates then will follow the boundary of the feasible set closely. Conversely, a large `TAU0` may degrade the reliability of the code.
Default: `TAU0` = 1.E0

DEL0 — In the initial phase of minimization a constraint is considered binding if

$$\frac{g_i(x)}{\max\left(1, \left\|\nabla g_i(x)\right\|\right)} \le \mathtt{DEL0} \qquad i = M_e + 1, \ldots, M$$

Good values are between .01 and 1.0. If `DEL0` is chosen too small then identification of the correct set of binding constraints may be delayed. Contrary, if `DEL0` is too large, then the method will often escape to the full regularized SQP method, using individual slack variables for any active constraint, which is quite costly. For well-scaled problems `DEL0=1.0` is reasonable. (Input)
Default: `DEL0` = .5*`TAU0`

SMALLW — Scalar containing the error allowed in the multipliers. For example, a negative multiplier of an inequality constraint is accepted (as zero) if its absolute value is less than `SMALLW`. (Input)
Default: `SMALLW` = exp(2*log(epsilon(x(1)/3)))

DELMIN — Scalar which defines allowable constraint violations of the final accepted result. Constraints are satisfied if $|g_i(x)| \le$ `DELMIN` , and $g_j(x) \ge$ (-`DELMIN`) respectively. (Input)
Default: `DELMIN =` min(`DEL0/10`, max(`EPSDIF`, min(`DEL0/10`, max(1.E-6*`DEL0`, `SMALLW`))

SCFMAX — Scalar containing the bound for the internal automatic scaling of the objective function. (Intput)
Default: `SCFMAX` = 1.0E4

FVALUE — Scalar containing the value of the objective function at the computed solution. (Output)

FORTRAN 90 Interface

Generic: `CALL NNLPG (FCN, GRAD, M, ME, IBTYPE, XLB, XUB, X [,…])`

Specific: The specific interface names are `S_NNLPG` and `D_NNLPG`.

Example 1

The problem

$$\begin{aligned} &\min F(x) = (x_1 - 2)^2 + (x_2 - 1)^2 \\ \text{subject to}\quad &g_1(x) = x_1 - 2x_2 + 1 = 0 \\ &g_2(x) = -x_1^2/4 - x_2^2 + 1 \ge 0 \end{aligned}$$

is solved.

```
      USE NNLPG_INT
      USE WRRRN_INT
      INTEGER    IBTYPE, M, ME
      PARAMETER  (IBTYPE=0, M=2, ME=1)
!
      REAL(KIND(1E0)) FVALUE, X(2), XGUESS(2), XLB(2), XUB(2)
      EXTERNAL FCN, GRAD
!
```

```
      XLB = -HUGE(X(1))
      XUB = HUGE(X(1))
!
      CALL NNLPG (FCN, GRAD, M, ME, IBTYPE, XLB, XUB, X)
!
      CALL WRRRN ('The solution is', X)
      END

      SUBROUTINE FCN (X, IACT, RESULT, IERR)
      INTEGER    IACT
      REAL(KIND(1E0)) X(*), RESULT
      LOGICAL IERR
!
      SELECT CASE (IACT)
      CASE(0)
         RESULT = (X(1)-2.0E0)**2 + (X(2)-1.0E0)**2
      CASE(1)
         RESULT = X(1) - 2.0E0*X(2) + 1.0E0
      CASE(2)
         RESULT = -(X(1)**2)/4.0E0 - X(2)**2 + 1.0E0
      END SELECT
      RETURN
      END

      SUBROUTINE GRAD (X, IACT, RESULT)
      INTEGER    IACT
      REAL(KIND(1E0)) X(*),RESULT(*)
!
      SELECT CASE (IACT)
      CASE(0)
         RESULT (1) = 2.0E0*(X(1)-2.0E0)
         RESULT (2) = 2.0E0*(X(2)-1.0E0)
      CASE(1)
         RESULT (1) = 1.0E0
         RESULT (2) = -2.0E0
      CASE(2)
         RESULT (1) = -0.5E0*X(1)
         RESULT (2) = -2.0E0*X(2)
      END SELECT
      RETURN
      END
```

Output

```
The solution is
1   0.8229
2   0.9114
```

Comments

1. Informational errors

Type	Code	
4	1	Constraint evaluation returns an error with current point.
4	2	Objective evaluation returns an error with current point.

4	3	Working set is singular in dual extended QP.
4	4	QP problem is seemingly infeasible.
4	5	A stationary point located.
4	6	A stationary point located or termination criteria too strong.
4	7	Maximum number of iterations exceeded.
4	8	Stationary point not feasible.
4	9	Very slow primal progress.
4	10	The problem is singular.
4	11	Matrix of gradients of binding constraints is singular or very ill-conditioned.
4	12	Small changes in the penalty function.

Description

The routine `NNLPG` provides an interface to a licensed version of subroutine `DONLP2`, a FORTRAN code developed by Peter Spellucci (1998). It uses a sequential equality constrained quadratic programming method with an active set technique, and an alternative usage of a fully regularized mixed constrained subproblem in case of nonregular constraints (i.e. linear dependent gradients in the "working sets"). It uses a slightly modified version of the Pantoja-Mayne update for the Hessian of the Lagrangian, variable dual scaling and an improved Armjijo-type stepsize algorithm. Bounds on the variables are treated in a gradient-projection like fashion. Details may be found in the following two papers:

P. Spellucci: *An SQP method for general nonlinear programs using only equality constrained subproblems*. Math. Prog. 82, (1998), 413-448.

P. Spellucci: *A new technique for inconsistent problems in the SQP method.* Math. Meth. of Oper. Res. 47, (1998), 355-500. (published by Physica Verlag, Heidelberg, Germany).

The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

$$\text{subject to} \qquad \begin{aligned} & g_j(x) = 0, \text{ for} && j = 1, \ldots, m_e \\ & g_j(x) \ge 0, \text{ for} && j = m_e + 1, \ldots, m \\ & x_l \le x \le x_u \end{aligned}$$

Although default values are provided for optional input arguments, it may be necessary to adjust these values for some problems. Through the use of optional arguments, NNLPG allows for several parameters of the algorithm to be adjusted to account for specific characteristics of problems. The DONLP2 Users Guide provides detailed descriptions of these parameters as well as strategies for maximizing the perfomance of the algorithm. The DONLP2 Users Guide is available in the "*help*" subdirectory of the main IMSL product installation directory. In addition, the following are a number of guidelines to consider when using NNLPG.

- A good initial starting point is very problem specific and should be provided by the calling program whenever possible. See optional argument `XGUESS`.

- If a two sided constraint $l_i \le g_i(x) \le u_i$ is transformed into two constraints $g_{2i}(x) \ge 0$ and $g_{2i+1}(x) \ge 0$, then choose $\mathrm{DEL0} < \frac{1}{2}(u_i - l_i) / max\{1, \|\nabla g_i(x)\|\}$, or at least try to provide an estimate for that value. This will increase the efficiency of the algorithm. See optional argument `DEL0`.
- The parameter `IERR` provided in the interface to the user supplied function `FCN` can be very useful in cases when evaluation is requested at a point that is not possible or reasonable. For example, if evaluation at the requested point would result in a floating point exception, then setting `IERR` to `.TRUE.` and returning without performing the evaluation will avoid the exception. NNLPG will then reduce the stepsize and try the step again. Note, if `IERR` is set to `.TRUE.` for the initial guess, then an error is issued.

Example 2

The same problem from Example 1 is solved, but here we use central differences to compute the gradient of the first constraint. This example demonstrates how `NNLPG` can be used in cases when analytic gradients are known for only a portion of the constraints and/or objective function. The subroutine `CDGRD` is used to compute an approximation to the gradient of the first constraint.

```
      USE NNLPG_INT
      USE CDGRD_INT
      USE WRRRN_INT
      INTEGER    IBTYPE, M, ME
      PARAMETER  (IBTYPE=0, M=2, ME=1)
!
      REAL(KIND(1E0)) FVALUE, X(2), XGUESS(2), XLB(2), XUB(2)
      EXTERNAL FCN, GRAD
!
      XLB = -HUGE(X(1))
      XUB = HUGE(X(1))
!
      CALL NNLPG (FCN, GRAD, M, ME, IBTYPE, XLB, XUB, X)
!
      CALL WRRRN ('The solution is', X)
      END

      SUBROUTINE FCN (X, IACT, RESULT, IERR)
      INTEGER    IACT
      REAL(KIND(1E0)) X(2), RESULT
      LOGICAL IERR
      EXTERNAL CONSTR1
!
      SELECT CASE (IACT)
      CASE(0)
         RESULT = (X(1)-2.0E0)**2 + (X(2)-1.0E0)**2
      CASE(1)
         CALL CONSTR1(2, X, RESULT)
      CASE(2)
         RESULT = -(X(1)**2)/4.0E0 - X(2)**2 + 1.0E0
      END SELECT
      RETURN
      END

      SUBROUTINE GRAD (X, IACT, RESULT)
      USE CDGRD_INT
      INTEGER    IACT
      REAL(KIND(1E0)) X(2),RESULT(2)
      EXTERNAL CONSTR1
!
      SELECT CASE (IACT)
      CASE(0)
         RESULT (1) = 2.0E0*(X(1)-2.0E0)
         RESULT (2) = 2.0E0*(X(2)-1.0E0)
      CASE(1)
         CALL CDGRD(CONSTR1, X, RESULT)
      CASE(2)
         RESULT (1) = -0.5E0*X(1)
         RESULT (2) = -2.0E0*X(2)
      END SELECT
      RETURN
      END

      SUBROUTINE CONSTR1 (N, X, RESULT)
      INTEGER N
      REAL(KIND(1E0)) X(*), RESULT
      RESULT = X(1) - 2.0E0*X(2) + 1.0E0
      RETURN
      END
```

Output

```
The solution is
1   0.8229
2   0.9114
```

CDGRD

Approximates the gradient using central differences.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function to be minimized. The usage is CALL FCN (N, X, F), where

N – Length of X. (Input)

X – The point at which the function is evaluated. (Input)
X should not be changed by FCN.

F – The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

XC — Vector of length N containing the point at which the gradient is to be estimated. (Input)

GC — Vector of length N containing the estimated gradient at XC. (Output)

Optional Arguments

N — Dimension of the problem. (Input)
Default: N = size (XC,1).

XSCALE — Vector of length N containing the diagonal scaling matrix for the variables. (Input)
In the absence of other information, set all entries to 1.0.
Default: XSCALE = 1.0.

EPSFCN — Estimate for the relative noise in the function. (Input)
EPSFCN must be less than or equal to 0.1. In the absence of other information, set EPSFCN to 0.0.
Default: EPSFCN = 0.0.

FORTRAN 90 Interface

Generic: CALL CDGRD (FCN, XC, GC [,…])

Specific: The specific interface names are S_CDGRD and D_CDGRD.

FORTRAN 77 Interface

Single: CALL CDGRD (FCN, N, XC, XSCALE, EPSFCN, GC)

Double: The double precision name is DCDGRD.

Example

In this example, the gradient of $f(x) = x_1 - x_1x_2 - 2$ is estimated by the finite-difference method at the point (1.0, 1.0).

```
      USE CDGRD_INT
      USE UMACH_INT
      INTEGER    I, N, NOUT
      PARAMETER  (N=2)
      REAL       EPSFCN, GC(N), XC(N)
      EXTERNAL   FCN
!                                  Initialization.
      DATA XC/2*1.0E0/
!                                  Set function noise.
      EPSFCN = 0.01
!
      CALL CDGRD (FCN, XC, GC, EPSFCN=EPSFCN)
```

```
!
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) (GC(I),I=1,N)
99999 FORMAT (’  The gradient is’, 2F8.2, /)
!
      END
!
      SUBROUTINE FCN (N, X, F)
      INTEGER    N
      REAL       X(N), F
!
      F = X(1) - X(1)*X(2) - 2.0E0
!
      RETURN
      END
```

Output

```
The gradient is    0.00   -1.00
```

Comments

This is Description A5.6.4, Dennis and Schnabel, 1983, page 323.

Description

The routine `CDGRD` uses the following finite-difference formula to estimate the gradient of a function of *n* variables at *x*:

$$\frac{f\left(x+h_i e_i\right)-f\left(x-h_i e_i\right)}{2h_i} \quad \text{for } i=1,\ldots,n$$

where $h_i = \varepsilon^{1/2} \max\{|x_i|, 1/s_i\} \operatorname{sign}(x_i)$, ε is the machine epsilon, s_i is the scaling factor of the *i*-th variable, and e_i is the *i*-th unit vector. For more details, see Dennis and Schnabel (1983).

Since the finite-difference method has truncation error, cancellation error, and rounding error, users should be aware of possible poor performance. When possible, high precision arithmetic is recommended.

FDGRD

Approximates the gradient using forward differences.

Required Arguments

FCN — User-supplied `SUBROUTINE` to evaluate the function to be minimized. The usage is `CALL FCN (N, X, F)`, where

N – Length of X. (Input)

X – The point at which the function is evaluated. (Input)
X should not be changed by FCN.

F – The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

XC — Vector of length N containing the point at which the gradient is to be estimated. (Input)

FC — Scalar containing the value of the function at XC. (Input)

GC — Vector of length N containing the estimated gradient at XC. (Output)

Optional Arguments

N — Dimension of the problem. (Input)
Default: N = size (XC,1).

XSCALE — Vector of length N containing the diagonal scaling matrix for the variables. (Input)
In the absence of other information, set all entries to 1.0.
Default: XSCALE = 1.0.

EPSFCN — Estimate of the relative noise in the function. (Input)
EPSFCN must be less than or equal to 0.1. In the absence of other information, set EPSFCN to 0.0.
Default: EPSFCN = 0.0.

FORTRAN 90 Interface

Generic: CALL FDGRD (FCN, XC, FC, GC [,…])

Specific: The specific interface names are S_FDGRD and D_FDGRD.

FORTRAN 77 Interface

Single: CALL FDGRD (FCN, XC, FC, GC, N, XSCALE, EPSFCN)

Double: The double precision name is DFDGRD.

Example

In this example, the gradient of $f(x) = x_1 - x_1 x_2 - 2$ is estimated by the finite-difference method at the point (1.0, 1.0).

```
USE FDGRD_INT
USE UMACH_INT
INTEGER    I, N, NOUT
PARAMETER  (N=2)
REAL       EPSFCN, FC, GC(N), XC(N)
```

```
      EXTERNAL   FCN
!                                    Initialization.
      DATA XC/2*1.0E0/
!                                    Set function noise.
      EPSFCN = 0.01
!                                    Get function value at current
!                                    point.
      CALL FCN (N, XC, FC)
!
      CALL FDGRD (FCN, XC, FC, GC, EPSFCN=EPSFCN)
!
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) (GC(I),I=1,N)
99999 FORMAT ('  The gradient is', 2F8.2, /)
!
      END
!
      SUBROUTINE FCN (N, X, F)
      INTEGER    N
      REAL       X(N), F
!
      F = X(1) - X(1)*X(2) - 2.0E0
!
      RETURN
      END
```

Output

```
The gradient is    0.00   -1.00
```

Comments

This is Description A5.6.3, Dennis and Schnabel, 1983, page 322.

Description

The routine FDGRD uses the following finite-difference formula to estimate the gradient of a function of n variables at x:

$$\frac{f\left(x+h_i e_i\right)-f\left(x\right)}{h_i} \quad \text{for } i=1,\ldots,n$$

where $h_i = \varepsilon^{1/2} \max\{|x_i|, 1/s_i\} \operatorname{sign}(x_i)$, ε is the machine epsilon, e_i is the i-th unit vector, and s_i is the scaling factor of the i-th variable. For more details, see Dennis and Schnabel (1983).

Since the finite-difference method has truncation error, cancellation error, and rounding error, users should be aware of possible poor performance. When possible, high precision arithmetic is recommended. When accuracy of the gradient is important, IMSL routine CDGRD (page 1336) should be used.

FDHES

Approximates the Hessian using forward differences and function values.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function to be minimized. The usage is CALL FCN (N, X, F), where

N – Length of X. (Input)

X – The point at which the function is evaluated. (Input)
X should not be changed by FCN.

F – The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

XC — Vector of length N containing the point at which the Hessian is to be approximated. (Input)

FC — Function value at XC. (Input)

H — N by N matrix containing the finite difference approximation to the Hessian in the lower triangle. (Output)

Optional Arguments

N — Dimension of the problem. (Input)
Default: N = size (XC,1).

XSCALE — Vector of length N containing the diagonal scaling matrix for the variables. (Input)
In the absence of other information, set all entries to 1.0.
Default: XSCALE = 1.0.

EPSFCN — Estimate of the relative noise in the function. (Input)
EPSFCN must be less than or equal to 0.1. In the absence of other information, set EPSFCN to 0.0.
Default: EPSFCN = 0.0.

LDH — Row dimension of H exactly as specified in the dimension statement of the calling program. (Input)
Default: LDH = size (H,1).

FORTRAN 90 Interface

Generic: CALL FDHES (FCN, XC, FC, H [,…])

Specific: The specific interface names are S_FDHES and D_FDHES.

FORTRAN 77 Interface

Single: `CALL FDHES (FCN, N, XC, XSCALE, FC, EPSFCN, H, LDH)`

Double: The double precision name is `DFDHES`.

Example

The Hessian is estimated for the following function at (1, −1)

$$f\left(x\right) = x_1^2 - x_1 x_2 - 2$$

```
      USE FDHES_INT
      USE UMACH_INT
!                                 Declaration of variables
      INTEGER    N, LDHES, NOUT
      PARAMETER  (N=2, LDHES=2)
      REAL       XC(N), FVALUE, HES(LDHES,N), EPSFCN
      EXTERNAL   FCN
!                                   Initialization
      DATA XC/1.0E0,-1.0E0/
!                                   Set function noise
      EPSFCN = 0.001
!                                   Evaluate the function at
!                                   current point
      CALL FCN (N, XC, FVALUE)
!                                 Get Hessian forward difference
!                                 approximation
      CALL FDHES (FCN, XC, FVALUE, HES, EPSFCN=EPSFCN)
!
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) ((HES(I,J),J=1,I),I=1,N)
99999 FORMAT ('  The lower triangle of the Hessian is', /,&
               5X,F10.2,/,5X,2F10.2,/)
!
      END
!
      SUBROUTINE FCN (N, X, F)
!                                  SPECIFICATIONS FOR ARGUMENTS
      INTEGER N
      REAL    X(N), F
!
      F = X(1)*(X(1) - X(2)) - 2.0E0
!
      RETURN
      END
```

Output

```
 The lower triangle of the Hessian is
  2.00
 -1.00      0.00
```

Comments

1. Workspace may be explicitly provided, if desired, by use of F2HES/DF2HES. The reference is:

```
CALL F2HES (FCN, N, XC, XSCALE, FC, EPSFCN, H, LDH, WK1, WK2)
```

The additional arguments are as follows:

WK1 — Real work vector of length N.

WK2 — Real work vector of length N.

2. This is Description A5.6.2 from Dennis and Schnabel, 1983; page 321.

Description

The routine FDHES uses the following finite-difference formula to estimate the Hessian matrix of function *f* at *x*:

$$\frac{f\left(x+h_i e_i+h_j e_j\right)-f\left(x+h_i e_i\right)-f\left(x+h_j e_j\right)+f\left(x\right)}{h_i h_j}$$

where $h_i = \varepsilon^{1/3} \max\{|x_i|, 1/s_i\} \operatorname{sign}(x_i)$, $h_j = \varepsilon^{1/3} \max\{|x_j|, 1/s_i\} \operatorname{sign}(x_j)$, ε is the machine epsilon or user-supplied estimate of the relative noise, s_i and s_j are the scaling factors of the *i*-th and *j*-th variables, and e_i and e_j are the *i*-th and *j*-th unit vectors, respectively. For more details, see Dennis and Schnabel (1983).

Since the finite-difference method has truncation error, cancellation error, and rounding error, users should be aware of possible poor performance. When possible, high precision arithmetic is recommended.

GDHES

Approximates the Hessian using forward differences and a user-supplied gradient.

Required Arguments

GRAD — User-supplied SUBROUTINE to compute the gradient at the point X. The usage is CALL GRAD (N, X, G), where

N – Length of X and G. (Input)

X – The point at which the gradient is evaluated. (Input)
X should not be changed by GRAD.

G – The gradient evaluated at the point X. (Output)

GRAD must be declared EXTERNAL in the calling program.

XC — Vector of length N containing the point at which the Hessian is to be estimated. (Input)

GC — Vector of length N containing the gradient of the function at XC. (Input)

H — N by N matrix containing the finite-difference approximation to the Hessian in the lower triangular part and diagonal. (Output)

Optional Arguments

N — Dimension of the problem. (Input)
Default: N = size (XC,1).

XSCALE — Vector of length N containing the diagonal scaling matrix for the variables. (Input)
In the absence of other information, set all entries to 1.0.
Default: XSCALE = 1.0.

EPSFCN — Estimate of the relative noise in the function. (Input)
EPSFCN must be less than or equal to 0.1. In the absence of other information, set EPSFCN to 0.0.
Default: EPSFCN = 0.0.

LDH — Leading dimension of H exactly as specified in the dimension statement of the calling program. (Input)
Default: LDH = size (H,1).

FORTRAN 90 Interface

Generic: CALL GDHES (GRAD, XC, GC, H [,…])

Specific: The specific interface names are S_GDHES and D_GDHES.

FORTRAN 77 Interface

Single: CALL GDHES (GRAD, N, XC, XSCALE, GC, EPSFCN, H, LDH)

Double: The double precision name is DGDHES.

Example

The Hessian is estimated by the finite-difference method at point (1.0, 1.0) from the following gradient functions:

$$g_1 = 2x_1x_2 - 2$$
$$g_2 = x_1x_1 + 1$$

```
      USE GDHES_INT
      USE UMACH_INT
!                                  Declaration of variables
      INTEGER    N, LDHES, NOUT
      PARAMETER  (N=2, LDHES=2)
      REAL       XC(N), GC(N), HES(LDHES,N)
      EXTERNAL   GRAD
!
      DATA XC/2*1.0E0/
!                                  Set function noise
!                                  Evaluate the gradient at the
!                                  current point
      CALL GRAD (N, XC, GC)
!                                  Get Hessian forward-difference
!                                  approximation
      CALL GDHES (GRAD, XC, GC, HES)
!
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) ((HES(I,J),J=1,N),I=1,N)
99999 FORMAT ('  THE HESSIAN IS', /, 2(5X,2F10.2,/),/)
!
      END
!
      SUBROUTINE GRAD (N, X, G)
!                                   SPECIFICATIONS FOR ARGUMENTS
      INTEGER N
      REAL    X(N), G(N)
!
      G(1) = 2.0E0*X(1)*X(2) - 2.0E0
      G(2) = X(1)*X(1) + 1.0E0
!
      RETURN
      END
```

Output

```
 THE HESSIAN IS
 2.00      2.00
 2.00      0.00
```

Comments

1. Workspace may be explicitly provided, if desired, by use of G2HES/DG2HES. The reference is:

   CALL G2HES (GRAD, N, XC, XSCALE, GC, EPSFCN, H, LDH, WK)

   The additional argument is

   WK — Work vector of length N.

2. This is Description A5.6.1, Dennis and Schnabel, 1983; page 320.

Description

The routine GDHES uses the following finite-difference formula to estimate the Hessian matrix of function F at x:

$$\frac{g\left(x+h_j e_j\right)-g\left(x\right)}{h_j}$$

where $h_j = \varepsilon^{1/2} \max\{|x_j|, 1/s_j\} \operatorname{sign}(x_j)$, ε is the machine epsilon, s_j is the scaling factor of the j-th variable, g is the analytic gradient of F at x, and e_j is the j-th unit vector. For more details, see Dennis and Schnabel (1983).

Since the finite-difference method has truncation error, cancellation error, and rounding error, users should be aware of possible poor performance. When possible, high precision arithmetic is recommended.

FDJAC

Approximates the Jacobian of M functions in N unknowns using forward differences.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function to be minimized. The usage is CALL FCN (M, N, X, F), where

M – Length of F. (Input)

N – Length of X. (Input)

X – The point at which the function is evaluated. (Input)
X should not be changed by FCN.

F – The computed function at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

XC — Vector of length N containing the point at which the gradient is to be estimated. (Input)

FC — Vector of length M containing the function values at XC. (Input)

FJAC — M by N matrix containing the estimated Jacobian at XC. (Output)

Optional Arguments

M — The number of functions. (Input)
Default: M = size (FC,1).

N — The number of variables. (Input)
Default: N = size (XC,1).

XSCALE — Vector of length N containing the diagonal scaling matrix for the variables. (Input)
In the absence of other information, set all entries to 1.0.
Default: XSCALE = 1.0.

EPSFCN — Estimate for the relative noise in the function. (Input)
EPSFCN must be less than or equal to 0.1. In the absence of other information, set EPSFCN to 0.0.
Default: EPSFCN = 0.0.

LDFJAC — Leading dimension of FJAC exactly as specified in the dimension statement of the calling program. (Input)
Default: LDFJAC = size (FJAC,1).

FORTRAN 90 Interface

Generic: CALL FDJAC (FCN, XC, FC, FJAC [,…])

Specific: The specific interface names are S_FDJAC and D_FDJAC.

FORTRAN 77 Interface

Single: CALL FDJAC (FCN, M, N, XC, XSCALE, FC, EPSFCN, FJAC, LDFJAC)

Double: The double precision name is DFDJAC.

Example

In this example, the Jacobian matrix of

$$f_1(x) = x_1 x_2 - 2$$
$$f_2(x) = x_1 - x_1 x_2 + 1$$

is estimated by the finite-difference method at the point (1.0, 1.0).

```
      USE FDJAC_INT
      USE UMACH_INT
!                                 Declaration of variables
      INTEGER   N, M, LDFJAC, NOUT
      PARAMETER (N=2, M=2, LDFJAC=2)
      REAL      FJAC(LDFJAC,N), XC(N), FC(M), EPSFCN
      EXTERNAL  FCN
!
      DATA XC/2*1.0E0/
!                                 Set function noise
      EPSFCN = 0.01
```

```
!                                 Evaluate the function at the
!                                 current point
      CALL FCN (M, N, XC, FC)
!                                 Get Jacobian forward-difference
!                                 approximation
      CALL FDJAC (FCN, XC, FC, FJAC, EPSFCN=EPFSCN)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) ((FJAC(I,J),J=1,N),I=1,M)
99999 FORMAT (’  The Jacobian is’, /, 2(5X,2F10.2,/),/)
!
      END
!
      SUBROUTINE FCN (M, N, X, F)
!                                  SPECIFICATIONS FOR ARGUMENTS
      INTEGER M, N
      REAL    X(N), F(M)
!
      F(1) = X(1)*X(2) - 2.0E0
      F(2) = X(1) - X(1)*X(2) + 1.0E0
!
      RETURN
      END
```

Output

```
 The Jacobian is
 1.00      1.00
 0.00     -1.00
```

Comments

1. Workspace may be explicitly provided, if desired, by use of F2JAC/DF2JAC. The reference is:

   CALL F2JAC (FCN, M, N, XC, XSCALE, FC, EPSFCN, FJAC, LDFJAC, WK)

   The additional argument is:

   WK — Work vector of length M.

2. This is Description A5.4.1, Dennis and Schnabel, 1983, page 314.

Description

The routine FDJAC uses the following finite-difference formula to estimate the Jacobian matrix of function f at x:

$$\frac{f\left(x+h_j e_j\right)-f\left(x\right)}{h_j}$$

where e_j is the j-th unit vector, $h_j = \varepsilon^{1/2} \max\{|x_j|, 1/s_j\} \operatorname{sign}(x_j)$, ε is the machine epsilon, and s_j is the scaling factor of the j-th variable. For more details, see Dennis and Schnabel (1983).

Since the finite-difference method has truncation error, cancellation error, and rounding error, users should be aware of possible poor performance. When possible, high precision arithmetic is recommended.

CHGRD

Checks a user-supplied gradient of a function.

Required Arguments

FCN — User-supplied `SUBROUTINE` to evaluate the function of which the gradient will be checked. The usage is `CALL FCN (N, X, F)`, where

`N` – Length of `X`. (Input)

`X` – The point at which the function is evaluated. (Input)
`X` should not be changed by `FCN`.

`F` – The computed function value at the point `X`. (Output)

`FCN` must be declared `EXTERNAL` in the calling program.

GRAD — Vector of length `N` containing the estimated gradient at `X`. (Input)

X — Vector of length `N` containing the point at which the gradient is to be checked. (Input)

INFO — Integer vector of length `N`. (Output)

`INFO`(`I`) = 0 means the user-supplied gradient is a poor estimate of the numerical gradient at the point `X`(`I`).

`INFO`(`I`) = 1 means the user-supplied gradient is a good estimate of the numerical gradient at the point `X`(`I`).

`INFO`(`I`) = 2 means the user-supplied gradient disagrees with the numerical gradient at the point `X`(`I`), but it might be impossible to calculate the numerical gradient.

`INFO`(`I`) = 3 means the user-supplied gradient and the numerical gradient are both zero at `X`(`I`), and, therefore, the gradient should be rechecked at a different point.

Optional Arguments

N — Dimension of the problem. (Input)
Default: `N` = size (`X`,1).

FORTRAN 90 Interface

Generic: `CALL CHGRD (FCN, GRAD, X, INFO [,…])`

Specific: The specific interface names are `S_CHGRD` and `D_CHGRD`.

FORTRAN 77 Interface

Single: `CALL CHGRD (FCN, GRAD, N, X, INFO)`

Double: The double precision name is `DCHGRD`.

Example

The user-supplied gradient of

$$f\left(x\right) = x_i + x_2 e^{-(t-x_3)2/x_4}$$

at (625, 1, 3.125, 0.25) is checked where $t = 2.125$.

```
      USE CHGRD_INT
      USE WRIRN_INT
!                                  Declare variables
      INTEGER    N
      PARAMETER  (N=4)
!
      INTEGER    INFO(N)
      REAL       GRAD(N), X(N)
      EXTERNAL   DRIV, FCN
!
!                                  Input values for point X
!                                  X = (625.0, 1.0, 3.125, .25)
!
      DATA X/625.0E0, 1.0E0, 3.125E0, 0.25E0/
!
      CALL DRIV (N, X, GRAD)
!
      CALL CHGRD (FCN, GRAD, X, INFO)
      CALL WRIRN ('The information vector', INFO, 1, N, 1)
!
      END
!
      SUBROUTINE FCN (N, X, FX)
      INTEGER    N
      REAL       X(N), FX
!
      REAL       EXP
      INTRINSIC  EXP
!
      FX = X(1) + X(2)*EXP(-1.0E0*(2.125E0-X(3))**2/X(4))
      RETURN
      END
!
      SUBROUTINE DRIV (N, X, GRAD)
      INTEGER    N
      REAL       X(N), GRAD(N)
!
      REAL       EXP
      INTRINSIC  EXP
!
      GRAD(1) = 1.0E0
      GRAD(2) = EXP(-1.0E0*(2.125E0-X(3))**2/X(4))
      GRAD(3) = X(2)*EXP(-1.0E0*(2.125E0-X(3))**2/X(4))*2.0E0/X(4)* &
               (2.125-X(3))
      GRAD(4) = X(2)*EXP(-1.0E0*(2.125E0-X(3))**2/X(4))* &
               (2.125E0-X(3))**2/(X(4)*X(4))
      RETURN
      END
```

Output

```
The information vector
1   2   3   4
1   1   1   1
```

Comments

1. Workspace may be explicitly provided, if desired, by use of C2GRD/DC2GRD. The reference is:

   CALL C2GRD (FCN, GRAD, N, X, INFO, FX, XSCALE, EPSFCN, XNEW)

   The additional arguments are as follows:

   FX — The functional value at X.

   XSCALE — Real vector of length N containing the diagonal scaling matrix.

   EPSFCN — The relative “noise” of the function FCN.

   XNEW — Real work vector of length N.

2. Informational errors

Type	Code	
4	1	The user-supplied gradient is a poor estimate of the numerical gradient.

Description

The routine CHGRD uses the following finite-difference formula to estimate the gradient of a function of n variables at x:

$$g_i(x) = \frac{f(x + h_i e_i) - f(x)}{h_i} \qquad \text{for } i=1, \ldots, n$$

where $h_i = \varepsilon^{1/2} \max\{|x_i|, 1/s_i\} \operatorname{sign}(x_i)$, ε is the machine epsilon, e_i is the *i*-th unit vector, and s_i is the scaling factor of the *i*-th variable.

The routine CHGRD checks the user-supplied gradient $\nabla f(x)$ by comparing it with the finite-difference gradient $g(x)$. If

$$\left|g_i(x)-(\nabla f(x))_i\right| < \tau\left|(\nabla f(x))_i\right|$$

where $\tau = \varepsilon^{1/4}$, then $(\nabla f(x))_i$, which is the *i*-th element of $\nabla f(x)$, is declared correct; otherwise, CHGRD computes the bounds of calculation error and approximation error. When both bounds are too small to account for the difference, $(\nabla f(x))_i$ is reported as incorrect. In the case of a large error bound, CHGRD uses a nearly optimal stepsize to recompute $g_i(x)$ and reports that $(\nabla f(x))_i$ is correct if

$$\left|g_i(x)-(\nabla f(x))_i\right| < 2\tau\left|(\nabla f(x))_i\right|$$

Otherwise, $(\nabla f(x))_i$ is considered incorrect unless the error bound for the optimal step is greater than $\tau\,|(\nabla f(x))_i|$. In this case, the numeric gradient may be impossible to compute correctly. For more details, see Schnabel (1985).

CHHES

Checks a user-supplied Hessian of an analytic function.

Required Arguments

GRAD — User-supplied SUBROUTINE to compute the gradient at the point X. The usage is CALL GRAD (N, X, G), where

N – Length of X and G.   (Input)

X – The point at which the gradient is evaluated. X should not be changed by GRAD. (Input)

G – The gradient evaluated at the point X.   (Output)

GRAD must be declared EXTERNAL in the calling program.

HESS — User-supplied SUBROUTINE to compute the Hessian at the point X. The usage is CALL HESS (N, X, H, LDH), where

N – Length of X.   (Input)

X – The point at which the Hessian is evaluated.   (Input)
X should not be changed by HESS.

H – The Hessian evaluated at the point X.   (Output)

LDH – Leading dimension of H exactly as specified in in the dimension statement of the calling program.   (Input)

HESS must be declared EXTERNAL in the calling program.

X — Vector of length N containing the point at which the Hessian is to be checked.   (Input)

INFO — Integer matrix of dimension N by N.   (Output)

INFO(I, J) = 0 means the Hessian is a poor estimate for function I at the point X(J).

INFO(I, J) = 1 means the Hessian is a good estimate for function I at the point X(J).

INFO(I, J) = 2 means the Hessian disagrees with the numerical Hessian for function I at the point X(J), but it might be impossible to calculate the numerical Hessian.

INFO(I, J) = 3 means the Hessian for function I at the point X(J) and the numerical Hessian are both zero, and, therefore, the gradient should be rechecked at a different point.

Optional Arguments

N — Dimension of the problem.   (Input)
Default: N = size (X,1).

LDINFO — Leading dimension of INFO exactly as specified in the dimension statement of the calling program.   (Input)
Default: LDINFO = size (INFO,1).

FORTRAN 90 Interface

Generic: CALL CHHES (GRAD, HESS, X, INFO [,…])

Specific: The specific interface names are S_CHHES and D_CHHES.

FORTRAN 77 Interface

Single: CALL CHHES (GRAD, HESS, N, X, INFO, LDINFO)

Double: The double precision name is DCHHES.

Example

The user-supplied Hessian of

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + \left(1 - x_1\right)^2$$

at (−1.2, 1.0) is checked, and the error is found.

```
      USE CHHES_INT
      INTEGER    LDINFO, N
      PARAMETER  (N=2, LDINFO=N)
!
      INTEGER    INFO(LDINFO,N)
      REAL       X(N)
      EXTERNAL   GRD, HES
!
!                                 Input values for X
!                                   X = (-1.2, 1.0)
!
      DATA X/-1.2, 1.0/
!
      CALL CHHES (GRD, HES, X, INFO)
!
      END
!
      SUBROUTINE GRD (N, X, UG)
      INTEGER    N
      REAL       X(N), UG(N)
!
      UG(1) = -400.0*X(1)*(X(2)-X(1)*X(1)) + 2.0*X(1) - 2.0
      UG(2) = 200.0*X(2) - 200.0*X(1)*X(1)
      RETURN
      END
!
      SUBROUTINE HES (N, X, HX, LDHS)
      INTEGER    N, LDHS
      REAL       X(N), HX(LDHS,N)
!
      HX(1,1) = -400.0*X(2) + 1200.0*X(1)*X(1) + 2.0
      HX(1,2) = -400.0*X(1)
      HX(2,1) = -400.0*X(1)
!                                  A sign change is made to HX(2,2)
!
      HX(2,2) = -200.0
      RETURN
      END
```

Output

```
*** FATAL    ERROR 1 from CHHES.  The Hessian evaluation with respect to
***          X(2) and X(2) is a poor estimate.
```

Comments

Workspace may be explicitly provided, if desired, by use of C2HES/DC2HES. The reference is

```
CALL C2HES (GRAD, HESS, N, X, INFO, LDINFO, G, HX, HS,
XSCALE, EPSFCN, INFT, NEWX)
```

The additional arguments are as follows:

G — Vector of length N containing the value of the gradient GRD at X.

HX — Real matrix of dimension N by N containing the Hessian evaluated at X.

HS — Real work vector of length N.

XSCALE — Vector of length N used to store the diagonal scaling matrix for the variables.

EPSFCN — Estimate of the relative noise in the function.

INFT — Vector of length N. For I = 1 through N, INFT contains information about the Jacobian.

NEWX — Real work array of length N.

Description

The routine CHHES uses the following finite-difference formula to estimate the Hessian of a function of n variables at x:

$$B_{ij}\left(x\right)=\left(g_i\left(x+h_je_j\right)-g_i\left(x\right)\right)/h_j \quad \text{for } j=1,\ldots,n$$

where $h_j = \varepsilon^{1/2}\max\{|x_j|, 1/s_j\}\ \text{sign}(x_j)$, ε is the machine epsilon, e_j is the j-th unit vector, s_j is the scaling factor of the j-th variable, and $g_i(x)$ is the gradient of the function with respect to the i-th variable.

Next, CHHES checks the user-supplied Hessian $H(x)$ by comparing it with the finite difference approximation $B(x)$. If

$$|B_{ij}(x) - H_{ij}(x)| < \tau\ |H_{ij}(x)|$$

where $\tau = \varepsilon^{1/4}$, then $H_{ij}(x)$ is declared correct; otherwise, CHHES computes the bounds of calculation error and approximation error. When both bounds are too small to account for the difference, $H_{ij}(x)$ is reported as incorrect. In the case of a large error bound, CHHES uses a nearly optimal stepsize to recompute $B_{ij}(x)$ and reports that $B_{ij}(x)$ is correct if

$$|B_{ij}(x) - H_{ij}(x)| < 2\tau\ |H_{ij}(x)|$$

Otherwise, $H_{ij}(x)$ is considered incorrect unless the error bound for the optimal step is greater than $\tau\ |H_{ij}(x)|$. In this case, the numeric approximation may be impossible to compute correctly. For more details, see Schnabel (1985).

CHJAC

Checks a user-supplied Jacobian of a system of equations with M functions in N unknowns.

Required Arguments

FCN — User-supplied SUBROUTINE to evaluate the function to be minimized. The usage is CALL FCN (M, N, X, F), where

M – Length of F. (Input)

N – Length of X. (Input)

X – The point at which the function is evaluated. (Input)
X should not be changed by FCN.

F – The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

JAC — User-supplied SUBROUTINE to evaluate the Jacobian at a point X. The usage is CALL JAC (M, N, X, FJAC, LDFJAC), where

M – Length of F. (Input)

N – Length of X. (Input)

X – The point at which the function is evaluated. (Input)
X should not be changed by FCN.

FJAC – The computed M by N Jacobian at the point X. (Output)

LDFJAC – Leading dimension of FJAC. (Input)

JAC must be declared EXTERNAL in the calling program.

X — Vector of length N containing the point at which the Jacobian is to be checked. (Input)

INFO — Integer matrix of dimension M by N. (Output)

INFO(I, J) = 0 means the user-supplied Jacobian is a poor estimate for function I at the point X(J).

INFO(I, J) = 1 means the user-supplied Jacobian is a good estimate for function I at the point X(J).

INFO(I, J) = 2 means the user-supplied Jacobian disagrees with the numerical Jacobian for function I at the point X(J), but it might be impossible to calculate the numerical Jacobian.

INFO(I, J) = 3 means the user-supplied Jacobian for function I at the point X(J) and the numerical Jacobian are both zero. Therefore, the gradient should be rechecked at a different point.

Optional Arguments

M — The number of functions in the system of equations. (Input)
Default: M = size (INFO,1).

N — The number of unknowns in the system of equations. (Input)
Default: N = size (X,1).

LDINFO — Leading dimension of INFO exactly as specified in the dimension statement of the calling program. (Input)
Default: LDINFO = size (INFO,1).

FORTRAN 90 Interface

Generic: CALL CHJAC (FCN, JAC, X, INFO [,…])

Specific: The specific interface names are S_CHJAC and D_CHJAC.

FORTRAN 77 Interface

Single: CALL CHJAC (FCN, JAC, M, N, X, INFO, LDINFO)

Double: The double precision name is DCHJAC.

Example

The user-supplied Jacobian of

$$f_1 = 1 - x_1$$
$$f_2 = 10\left(x_2 - x_1^2\right)$$

at (−1.2, 1.0) is checked.

```
      USE CHJAC_INT
      USE WRIRN_INT
      INTEGER    LDINFO, N
      PARAMETER  (M=2,N=2,LDINFO=M)
!
      INTEGER    INFO(LDINFO,N)
      REAL       X(N)
      EXTERNAL   FCN, JAC
!
!                                 Input value for X
!                                    X = (-1.2, 1.0)
!
      DATA X/-1.2, 1.0/
!
      CALL CHJAC (FCN, JAC, X, INFO)
      CALL WRIRN ('The information matrix', INFO)
!
      END
```

```
!
      SUBROUTINE FCN (M, N, X, F)
      INTEGER    M, N
      REAL       X(N), F(M)
!
      F(1) = 1.0 - X(1)
      F(2) = 10.0*(X(2)-X(1)*X(1))
      RETURN
      END
!
      SUBROUTINE JAC (M, N, X, FJAC, LDFJAC)
      INTEGER    M, N, LDFJAC
      REAL       X(N), FJAC(LDFJAC,N)
!
      FJAC(1,1) = -1.0
      FJAC(1,2) = 0.0
      FJAC(2,1) = -20.0*X(1)
      FJAC(2,2) = 10.0
      RETURN
      END
```

Output

```
*** WARNING  ERROR 2 from C2JAC.  The numerical value of the Jacobian
***          evaluation for function 1 at the point X(2) = 1.000000E+00 and
***          the user-supplied value are both zero.  The Jacobian for this
***          function should probably be re-checked at another value for
***          this point.

The information matrix
    1   2
1   1   3
2   1   1
```

Comments

1. Workspace may be explicitly provided, if desired, by use of C2JAC/DC2JAC. The reference is:

   ```
   CALL C2JAC (FCN, JAC, N, X, INFO, LDINFO, FX, FJAC,
   GRAD, XSCALE, EPSFCN, INFT, NEWX)
   ```

   The additional arguments are as follows:

   FX — Vector of length M containing the value of each function in FCN at X.

   FJAC — Real matrix of dimension M by N containing the Jacobian of FCN evaluated at X.

   GRAD — Real work vector of length N used to store the gradient of each function in FCN.

   XSCALE — Vector of length N used to store the diagonal scaling matrix for the variables.

EPSFCN — Estimate of the relative noise in the function.

INFT — Vector of length `N`. For I = 1 through `N`, `INFT` contains information about the Jacobian.

NEWX — Real work array of length `N`.

2. Informational errors

Type	Code	
4	1	The user-supplied Jacobian is a poor estimate of the numerical Jacobian.

Description

The routine `CHJAC` uses the following finite-difference formula to estimate the gradient of the i-th function of n variables at x:

$$g_{ij}(x) = (f_i(x + h_je_j) - f_i(x))/h_j \quad \text{for } j = 1, \ldots, n$$

where $h_j = \varepsilon^{1/2}\max\{|x_j|, 1/s_j\}\ \text{sign}(x_j)$, ε is the machine epsilon, e_j is the j-th unit vector, and s_j is the scaling factor of the j-th variable.

Next, `CHJAC` checks the user-supplied Jacobian $J(x)$ by comparing it with the finite difference gradient $g_i(x)$. If

$$|g_{ij}(x) - J_{ij}(x)| < \tau\ |J_{ij}(x)|$$

where $\tau = \varepsilon^{1/4}$, then $\mathrm{J}_{ij}(x)$ is declared correct; otherwise, `CHJAC` computes the bounds of calculation error and approximation error. When both bounds are too small to account for the difference, $J_{ij}(x)$ is reported as incorrect. In the case of a large error bound, `CHJAC` uses a nearly optimal stepsize to recompute $g_{ij}(x)$ and reports that $J_{ij}(x)$ is correct if

$$|g_{ij}(x) - J_{ij}(x)| < 2\tau\ |J_{ij}(x)|$$

Otherwise, $J_{ij}(x)$ is considered incorrect unless the error bound for the optimal step is greater than $\tau\ |J_{ij}(x)|$. In this case, the numeric gradient may be impossible to compute correctly. For more details, see Schnabel (1985).

GGUES

Generates points in an `N`-dimensional space.

Required Arguments

A — Vector of length `N`.   (Input)
See `B`.

B — Real vector of length N. (Input)
A and B define the rectangular region in which the points will be generated, i.e., A(I) < S(I) < B(I) for I = 1, 2, …, N. Note that if B(I) < A(I), then B(I) < S(I) < A(I).

K — The number of points to be generated. (Input)

IDO — Initialization parameter. (Input/Output)
IDO must be set to zero for the first call. GGUES resets IDO to 1 and returns the first generated point in S. Subsequent calls should be made with IDO = 1.

S — Vector of length N containing the generated point. (Output)
Each call results in the next generated point being stored in S.

Optional Arguments

N — Dimension of the space. (Input)
Default: N = size (B,1).

FORTRAN 90 Interface

Generic: CALL GGUES (A, B, K, IDO, S [,…])

Specific: The specific interface names are S_GGUES and D_GGUES.

FORTRAN 77 Interface

Single: CALL GGUES (N, A, B, K, IDO, S)

Double: The double precision name is DGGUES.

Example

We want to search the rectangle with vertices at coordinates (1, 1), (3, 1), (3, 2), and (1, 2) ten times for a global optimum of a nonlinear function. To do this, we need to generate starting points. The following example illustrates the use of GGUES in this process:

```
      USE GGUES_INT
      USE UMACH_INT
!                                 Variable Declarations
      INTEGER    N
      PARAMETER  (N=2)
!
      INTEGER    IDO, J, K, NOUT
      REAL       A(N), B(N), S(N)
!                                 Initializations
!
!                                 A   = ( 1.0, 1.0)
!                                 B   = ( 3.0, 2.0)
!
      DATA A/1.0, 1.0/
      DATA B/3.0, 2.0/
!
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99998)
99998 FORMAT (’  Point Number’, 7X, ’Generated Point’)
!
      K = 10
      IDO = 0
      DO 10  J=1, K
         CALL GGUES (A, B, K, IDO, S)
!
         WRITE (NOUT,99999) J, S(1), S(2)
99999    FORMAT (1X, I7, 14X, ’(’, F4.1, ’,’, F6.3, ’)’)
!
   10 CONTINUE
!
      END
```

Output

```
 Point Number       Generated Point
       1              ( 1.5, 1.125)
       2              ( 2.0, 1.500)
       3              ( 2.5, 1.750)
       4              ( 1.5, 1.375)
       5              ( 2.0, 1.750)
       6              ( 1.5, 1.625)
       7              ( 2.5, 1.250)
       8              ( 1.5, 1.875)
       9              ( 2.0, 1.250)
      10              ( 2.5, 1.500)
```

Comments

1. Workspace may be explicitly provided, if desired, by use of G2UES/DG2UES. The reference is:

   ```
   CALL G2UES (N, A, B, K, IDO, S, WK, IWK)
   ```

   The additional arguments are:

   WK — Work vector of length N. WK must be preserved between calls to G2UES.

   IWK — Work vector of length 10. IWK must be preserved between calls to G2UES.

2. Informational error

Type	Code	
4	1	Attempt to generate more than K points.

3. The routine GGUES may be used with any nonlinear optimization routine that requires starting points. The rectangle to be searched (defined by A, B, and N) must be determined; and the number of starting points, K, must be chosen. One possible use for

GGUES would be to call GGUES to generate a point in the chosen rectangle. Then, call the nonlinear optimization routine using this point as an initial guess for the solution. Repeat this process K times. The number of iterations that the optimization routine is allowed to perform should be quite small (5 to 10) during this search process. The best (or best several) point(s) found during the search may be used as an initial guess to allow the optimization routine to determine the optimum more accurately. In this manner, an N dimensional rectangle may be effectively searched for a global optimum of a nonlinear function. The choice of K depends upon the nonlinearity of the function being optimized. A function with many local optima requires a larger value than a function with only a few local optima.

Description

The routine GGUES generates starting points for algorithms that optimize functions of several variables–or, almost equivalently–algorithms that solve simultaneous nonlinear equations.

The routine GGUES is based on systematic placement of points to optimize the dispersion of the set. For more details, see Aird and Rice (1977).

Chapter 9: Basic Matrix/Vector Operations

Routines

9.1. Basic Linear Algebra Subprograms (BLAS)

9.2. Other Matrix/Vector Operations

9.2.1 Matrix Copy

9.2.2 Matrix Conversion

Basic Linear Algebra Subprograms

The basic linear algebra subprograms, normally referred to as the BLAS, are routines for low-level operations such as dot products, matrix times vector, and matrix times matrix. Lawson et al. (1979) published the original set of 38 BLAS. The IMSL BLAS collection includes these 38 subprograms plus additional ones that extend their functionality. Since Dongarra et al. (1988 and 1990) published extensions to this set, it is customary to refer to the original 38 as Level 1 BLAS. The Level 1 operations are performed on one or two vectors of data. An extended set of subprograms perform operations involving a matrix and one or two vectors. These are called the Level 2 BLAS (page 1377). An additional extended set of operations on matrices is called the Level 3 BLAS (page 1377).

Users of the BLAS will often benefit from using versions of the BLAS supplied by hardware vendors, if available. This can provide for more efficient execution of many application programs. The BLAS provided by IMSL are written in FORTRAN. Those supplied by vendors may be written in other languages, such as assembler. The documentation given below for the BLAS is compatible with a vendor's version of the BLAS that conforms to the published specifications.

Programming Notes for Level 1 BLAS

The Level 1 BLAS do not follow the usual IMSL naming conventions. Instead, the names consist of a prefix of one or more of the letters "I," "S," "D," "C" and "Z;" a root name; and sometimes a suffix. For subprograms involving a mixture of data types, the output type is indicated by the first prefix letter. The suffix denotes a variant algorithm. The prefix denotes the type of the operation according to the following table:

I	Integer		
S	Real	C	Complex
D	Double	Z	Double complex
SD	Single and Double	CZ	Single and double complex
DQ	Double and Quadruple	ZQ	Double and quadruple complex

Vector arguments have an increment parameter that specifies the storage space or stride between elements. The correspondence between the vectors x and y and the arguments SX and SY, and INCX and INCY is

$$x_i = \begin{cases} \mathrm{SX}\left(\left(\mathrm{I\text{-}1}\right) * \mathrm{INCX} + 1\right) & \text{if INCX} \ge 0 \\ \mathrm{SX}\left(\left(\mathrm{I\text{-}N}\right) * \mathrm{INCX} + 1\right) & \text{if INCX} < 0 \end{cases}$$

$$y_i = \begin{cases} \mathrm{SY}\left(\left(\mathrm{I\text{-}1}\right) * \mathrm{INCY} + 1\right) & \text{if INCY} \ge 0 \\ \mathrm{SY}\left(\left(\mathrm{I\text{-}N}\right) * \mathrm{INCY} + 1\right) & \text{if INCY} < 0 \end{cases}$$

Function subprograms SXYZ, DXYZ, page 1372, refer to a third vector argument z. The storage increment INCZ for z is defined like INCX, INCY. In the Level 1 BLAS, only positive values of INCX are allowed for operations that have a single vector argument. The loops in all of the Level 1 BLAS process the vector arguments in order of increasing i. For INCX, INCY, INCZ < 0, this implies processing in reverse storage order.

The function subprograms in the Level 1 BLAS are all illustrated by means of an assignment statement. For example, see SDOT (page 1370). Any value of a function subprogram can be used in an expression or as a parameter passed to a subprogram as long as the data types agree.

Descriptions of the Level 1 BLAS Subprograms

The set of Level 1 BLAS are summarized in Table 9.1. This table also lists the page numbers where the subprograms are described in more detail.

Specification of the Level 1 BLAS

With the definitions,

$$\mathrm{MX} = \max\,\{1, 1 + (N - 1)|\mathrm{INCX}|\}$$

$$\mathrm{MY} = \max\,\{1, 1 + (N - 1)|\mathrm{INCY}|\}$$

$$\mathrm{MZ} = \max\,\{1, 1 + (N - 1)|\mathrm{INCZ}|\}$$

the subprogram descriptions assume the following FORTRAN declarations:

```
IMPLICIT INTEGER          (I-N)
IMPLICIT REAL             S
IMPLICIT DOUBLE PRECISION D
IMPLICIT COMPLEX          C
IMPLICIT DOUBLE COMPLEX   Z

INTEGER                   IX(MX)
REAL                      SX(MX), SY(MY), SZ(MZ),
                          SPARAM(5)
DOUBLE PRECISION          DX(MX), DY(MY), DZ(MZ),
                          DPARAM(5)
DOUBLE PRECISION          DACC(2), DZACC(4)
COMPLEX                   CX(MX), CY(MY)
DOUBLE COMPLEX            ZX(MX), ZY(MY)
```

Since FORTRAN 77 does not include the type DOUBLE COMPLEX, subprograms with DOUBLE COMPLEX arguments are not available for all systems. Some systems use the declaration COMPLEX * 16 instead of DOUBLE COMPLEX.

In the following descriptions, the original BLAS are marked with an * in the left column.

Table 9.1: Level 1 Basic Linear Algebra Subprograms

Operation	Integer	Real	Double	Complex	Double Complex	Pg.
$x_i \leftarrow a$	ISET	SSET	DSET	CSET	ZSET	1369
$y_i \leftarrow x_i$	ICOPY	SCOPY	DCOPY	CCOPY	ZCOPY	1369
$x_i \leftarrow ax_i$ $a \in \mathbf{R}$		SSCAL	DSCAL	CSCAL CSSCAL	ZSCAL ZDSCAL	1369

Operation	Integer	Real	Double	Complex	Double Complex	Pg.
$y_i \leftarrow ax_i$ $a \in \mathbf{R}$		SVCAL	DVCAL	CVCAL CSVCAL	ZVCAL ZDVCAL	1369
$x_i \leftarrow x_i + a$	IADD	SADD	DADD	CADD	ZADD	1370
$x_i \leftarrow a - x_i$	ISUB	SSUB	DSUB	CSUB	ZSUB	1370
$y_i \leftarrow ax_i + y_i$		SAXPY	DAXPY	CAXPY	ZAXPY	1370
$y_i \leftrightarrow x_i$	ISWAP	SSWAP	DSWAP	CSWAP	ZSWAP	1370
$x \cdot y$ $\bar{x} \cdot \mathrm{y}$		SDOT	DDOT	CDOTU CDOTC	ZDOTU ZDOTC	1370
$x \cdot y \dagger$ $\bar{x} \cdot y \dagger$		DSDOT		CZDOTU CZDOTC	ZQDOTU ZQDOTC	1371
$a + x \cdot y \dagger$ $a + \bar{x} \cdot y \dagger$		SDSDOT	DQDDOT	CZUDOT CZCDOT	ZQUDOT ZQCDOT	1371
$b + x \cdot y \dagger$ $\mathrm{ACC} + b + x \cdot y \dagger$		SDDOTI SDDOTA	DQDOTI DQDOTA	CZDOTI CZDOTA	ZQDOTI ZQDOTA	1372
$z_i \leftarrow x_i y_i$		SHPROD	DHPROD			1372
$\sum x_i y_i z_i$		SXYZ	DXYZ			1372
$\sum x_i$	ISUM	SSUM	DSUM			1372
$\sum \lvert x_i \rvert$		SASUM	DASUM	SCASUM	DZASUM	1373
$\lVert x \rVert_2$		SNRM2	DNRM2	SCNRM2	DZNRM2	1373
$\Pi\, x_i$		SPRDCT	DPRDCT			1373
$i:\ x_i = \min_j x_j$	IIMIN	ISMIN	IDMIN			1374
$i:\ x_i = \max_j x_j$	IIMAX	ISMAX	IDMAX			1374
$i:\ \lvert x_i \rvert = \min_j \lvert x_j \rvert$		ISAMIN	IDAMIN	ICAMIN	IZAMIN	1374
$i:\ \lvert x_i \rvert = \max_j \lvert x_j \rvert$		ISAMAX	IDAMAX	ICAMAX	IZAMAX	1374
Construct Givens rotation		SROTG	DROTG			1374
Apply Givens rotation		SROT	DROT	CSROT	ZDROT	1375

Operation	Integer	Real	Double	Complex	Double Complex	Pg.
Construct modified Givens transform		SROTMG	DROTMG			1376
Apply modified Givens transform		SROTM	DROTM	CSROTM	ZDROTM	1377

†Higher precision accumulation used

Set a Vector to a Constant Value

```
CALL ISET (N, IA, IX, INCX)
CALL SSET (N, SA, SX, INCX)
CALL DSET (N, DA, DX, INCX)
CALL CSET (N, CA, CX, INCX)
CALL ZSET (N, ZA, ZX, INCX)
```

These subprograms set $x_i \leftarrow a$ for $i = 1, 2, \ldots, N$. If $N \le 0$, then the subprograms return immediately.

Copy a Vector

```
 CALL ICOPY (N, IX, INCX, IY, INCY)
*CALL SCOPY (N, SX, INCX, SY, INCY)
*CALL DCOPY (N, DX, INCX, DY, INCY)
*CALL CCOPY (N, CX, INCX, CY, INCY)
 CALL ZCOPY (N, ZX, INCX, ZY, INCY)
```

These subprograms set $y_i \leftarrow x_i$ for $i = 1, 2, \ldots, N$. If $N \le 0$, then the subprograms return immediately.

Scale a Vector

```
*CALL SSCAL (N, SA, SX, INCX)
*CALL DSCAL (N, DA, DX, INCX)
*CALL CSCAL (N, CA, CX, INCX)
 CALL ZSCAL (N, ZA, ZX, INCX)
*CALL CSSCAL (N, SA, CX, INCX)
 CALL ZDSCAL (N, DA, ZX, INCX)
```

These subprograms set $x_i \leftarrow ax_i$ for $i = 1, 2, \ldots, N$. If $N \le 0$, then the subprograms return immediately. CAUTION: For `CSSCAL` and `ZDSCAL`, the scalar quantity a is real and the vector x is complex.

Multiply a Vector by a Constant

```
CALL SVCAL (N, SA, SX, INCX, SY, INCY)
CALL DVCAL (N, DA, DX, INCX, DY, INCY)
CALL CVCAL (N, CA, CX, INCX, CY, INCY)
CALL ZVCAL (N, ZA, ZX, INCX, ZY, INCY)
CALL CSVCAL (N, SA, CX, INCX, CY, INCY)
CALL ZDVCAL (N, DA, ZX, INCX, ZY, INCY)
```

These subprograms set $y_i \leftarrow ax_i$ for $i = 1, 2, \ldots, N$. If $N \le 0$, then the subprograms return immediately. CAUTION: For `CSVCAL` and `ZDVCAL`, the scalar quantity a is real and the vector x is complex.

Add a Constant to a Vector

```
CALL IADD (N, IA, IX, INCX)
CALL SADD (N, SA, SX, INCX)
CALL DADD (N, DA, DX, INCX)
CALL CADD (N, CA, CX, INCX)
CALL ZADD (N, ZA, ZX, INCX)
```

These subprograms set $x_i \leftarrow x_i + a$ for $i = 1, 2, \ldots, N$. If $N \le 0$, then the subprograms return immediately.

Subtract a Vector from a Constant

```
CALL ISUB (N, IA, IX, INCX)
CALL SSUB (N, SA, SX, INCX)
CALL DSUB (N, DA, DX, INCX)
CALL CSUB (N, CA, CX, INCX)
CALL ZSUB (N, ZA, ZX, INCX)
```

These subprograms set $x_i \leftarrow a - x_i$ for $i = 1, 2, \ldots, N$. If $N \le 0$, then the subprograms return immediately.

Constant Times a Vector Plus a Vector

```
*CALL SAXPY (N, SA, SX, INCX, SY, INCY)
*CALL DAXPY (N, DA, DX, INCX, DY, INCY)
*CALL CAXPY (N, CA, CX, INCX, CY, INCY)
 CALL ZAXPY (N, ZA, ZX, INCX, ZY, INCY)
```

These subprograms set $y_i \leftarrow ax_i + y_i$ for $i = 1, 2, \ldots, N$. If $N \le 0$, then the subprograms return immediately.

Swap Two Vectors

```
 CALL ISWAP (N, IX, INCX, IY, INCY)
*CALL SSWAP (N, SX, INCX, SY, INCY)
*CALL DSWAP (N, DX, INCX, DY, INCY)
*CALL CSWAP (N, CX, INCX, CY, INCY)
 CALL ZSWAP (N, ZX, INCX, ZY, INCY)
```

These subprograms perform the exchange $y_i \leftrightarrow x_i$ for $i = 1, 2, \ldots, N$. If $N \le 0$, then the subprograms return immediately.

Dot Product

```
*SW =  SDOT  (N, SX, INCX, SY, INCY)
*DW =  DDOT  (N, DX, INCX, DY, INCY)
```

```
*CW =  CDOTU (N, CX, INCX, CY, INCY)
*CW =  CDOTC (N, CX, INCX, CY, INCY)
 ZW =  ZDOTU (N, ZX, INCX, ZY, INCY)
 ZW =  ZDOTC (N, ZX, INCX, ZY, INCY)
```

The function subprograms SDOT, DDOT, CDOTU, and ZDOTU compute

$$\sum_{i=1}^{N} x_i y_i$$

The function subprograms CDOTC and ZDOTC compute

$$\sum_{i=1}^{N} \bar{x}_i y_i$$

The suffix C indicates that the complex conjugates of x_i are used. The suffix U indicates that the unconjugated values of x_i are used. If $N \le 0$, then the subprograms return zero.

Dot Product with Higher Precision Accumulation

```
*DW =  DSDOT  (N, SX, INCX, SY, INCY)
 CW =  CZDOTC (N, CX, INCX, CY, INCY)
 CW =  CZDOTU (N, CX, INCX, CY, INCY)
 ZW =  ZQDOTC (N, ZX, INCX, ZY, INCY)
 ZW =  ZQDOTU (N, ZX, INCX, ZY, INCY)
```

The function subprogram DSDOT computes

$$\sum_{i=1}^{N} x_i y_i$$

using double precision accumulation. The function subprograms CZDOTU and ZQDOTU compute

$$\sum_{i=1}^{N} x_i y_i$$

using double and quadruple complex accumulation, respectively. The function subprograms CZDOTC and ZQDOTC compute

$$\sum_{i=1}^{N} \bar{x}_i y_i$$

using double and quadruple complex accumulation, respectively. If $N \le 0$, then the subprograms return zero.

Constant Plus Dot Product with Higher Precision Accumulation

```
*SW = SDSDOT (N, SA, SX, INCX, SY, INCY)
 DW = DQDDOT (N, DA, DX, INCX, DY, INCY)
 CW = CZCDOT (N, CA, CX, INCX, CY, INCY)
 CW = CZUDOT (N, CA, CX, INCX, CY, INCY)
 ZW = ZQCDOT (N, ZA, ZX, INCX, ZY, INCY)
 ZW = ZQUDOT (N, ZA, ZX, INCX, ZY, INCY)
```

The function subprograms SDSDOT, DQDDOT, CZUDOT, and ZQUDOT compute

$$a + \sum_{i=1}^{N} x_i y_i$$

using higher precision accumulation where SDSDOT uses double precision accumulation, DQDDOT uses quadruple precision accumulation, CZUDOT uses double complex accumulation, and ZQUDOT uses quadruple complex accumulation. The function subprograms CZCDOT and ZQCDOT compute

$$a + \sum_{i=1}^{N} \bar{x}_i y_i$$

using double complex and quadruple complex accumulation, respectively. If $N \le 0$, then the subprograms return zero.

Dot Product Using the Accumulator

```
 SW =  SDDOTI (N, SB,  DACC, SX, INCX, SY, INCY)
 SW =  SDDOTA (N, SB,  DACC, SX, INCX, SY, INCY)
 CW =  CZDOTI (N, CB,  DACC, CX, INCX, CY, INCY)
 CW =  CZDOTA (N, CB,  DACC, CX, INCX, CY, INCY)
*DW =  DQDOTI (N, DB,  DACC, DX, INCX, DY, INCY)
*DW =  DQDOTA (N, DB,  DACC, DX, INCX, DY, INCY)
 ZW =  ZQDOTI (N, ZB, DZACC, ZX, INCX, ZY, INCY)
 ZW =  ZQDOTA (N, ZB, DZACC, ZX, INCX, ZY, INCY)
```

The variable DACC, a double precision array of length two, is used as a quadruple precision accumulator. DZACC, a double precision array of length four, is its complex analog. The function subprograms, with a name ending in I, initialize DACC to zero. All of the function subprograms then compute

$$\mathrm{DACC} + b + \sum_{i=1}^{N} x_i y_i$$

and store the result in DACC. The result, converted to the precision of the function, is also returned as the function value. If $N \le 0$, then the function subprograms return zero.

Hadamard Product

```
CALL SHPROD (N, SX, INCX, SY, INCY, SZ, INCZ)
CALL DHPROD (N, DX, INCX, DY, INCY, DZ, INCZ)
```

These subprograms set $z_i \leftarrow x_i y_i$ for $i = 1, 2, \ldots, N$. If $N \le 0$, then the subprograms return immediately.

Triple Inner Product

```
SW = SXYZ (N, SX, INCX, SY, INCY, SZ, INCZ)
DW = DXYZ (N, DX, INCX, DY, INCY, DZ, INCZ)
```

These function subprograms compute

$$\sum_{i=1}^{N} x_i y_i z_i$$

If $N \le 0$ then the subprograms return zero.

Sum of the Elements of a Vector

```
IW = ISUM (N, IX, INCX)
SW = SSUM (N, SX, INCX)
DW = DSUM (N, DX, INCX)
```

These function subprograms compute

$$\sum_{i=1}^{N} x_i$$

If $N \le 0$, then the subprograms return zero.

Sum of the Absolute Values of the Elements of a Vector

```
*SW = SASUM (N, SX, INCX)
*DW = DASUM (N, DX, INCX)
*SW = SCASUM (N, CX, INCX)
 DW = DZASUM (N, ZX, INCX)
```

The function subprograms `SASUM` and `DASUM` compute

$$\sum_{i=1}^{N} \left| x_i \right|$$

The function subprograms `SCASUM` and `DZASUM` compute

$$\sum_{i=1}^{N} \left[\left| \Re x_i \right| + \left| \Im x_i \right| \right]$$

If $N \le 0$, then the subprograms return zero. CAUTION: For `SCASUM` and `DZASUM`, the function subprogram returns a real value.

Euclidean or ℓ_2 Norm of a Vector

```
*SW = SNRM2  (N, SX, INCX)
*DW = DNRM2  (N, DX, INCX)
*SW = SCNRM2 (N, CX, INCX)
 DW = DZNRM2 (N, ZX, INCX)
```

These function subprograms compute

$$\left[\sum_{i=1}^{N} \left| x_i \right|^2 \right]^{1/2}$$

If $N \le 0$, then the subprograms return zero. CAUTION: For `SCNRM2` and `DZNRM2`, the function subprogram returns a real value.

Product of the Elements of a Vector

```
SW = SPRDCT (N, SX, INCX)
DW = DPRDCT (N, DX, INCX)
```

These function subprograms compute

$$\prod_{i=1}^{N} x_i$$

If $N \le 0$, then the subprograms return zero.

Index of Element Having Minimum Value

```
IW = IIMIN (N, IX, INCX)
IW = ISMIN (N, SX, INCX)
IW = IDMIN (N, DX, INCX)
```

These function subprograms compute the smallest index i such that $x_i = \min_{1 \le j \le N} x_j$. If $N \le 0$, then the subprograms return zero.

Index of Element Having Maximum Value

```
IW = IIMAX (N, IX, INCX)
IW = ISMAX (N, SX, INCX)
IW = IDMAX (N, DX, INCX)
```

These function subprograms compute the smallest index i such that$x_i = \max_{1 \le j \le N} x_j$. If $N \le 0$, then the subprograms return zero.

Index of Element Having Minimum Absolute Value

```
IW = ISAMIN (N, SX, INCX)
IW = IDAMIN (N, DX, INCX)
IW = ICAMIN (N, CX, INCX)
IW = IZAMIN (N, ZX, INCX)
```

The function subprograms `ISAMIN` and `IDAMIN` compute the smallest index i such that $|x_i| = \min_{1 \le j \le N} |x_j|$. The function subprograms `ICAMIN` and `IZAMIN` compute the smallest index i such that

$$\left|\Re x_i\right| + \left|\Im x_i\right| = \min_{1 \le j \le N} \left[\left|\Re x_j\right| + \left|\Im x_j\right|\right]$$

If $N \le 0$, then the subprograms return zero.

Index of Element Having Maximum Absolute Value

```
*IW = ISAMAX (N, SX, INCX)
*IW = IDAMAX (N, DX, INCX)
*IW = ICAMAX (N, CX, INCX)
 IW = IZAMAX (N, ZX, INCX)
```

The function subprograms `ISAMAX` and `IDAMAX` compute the smallest index i such that $|x_i| = \max_{1 \le j \le N} |x_j|$. The function subprograms `ICAMAX` and `IZAMAX` compute the smallest index i such that

$$\left|\Re x_i\right| + \left|\Im x_i\right| = \max_{1 \le j \le N} \left[\left|\Re x_j\right| + \left|\Im x_j\right|\right]$$

If $N \le 0$, then the subprograms return zero.

Construct a Givens Plane Rotation

```
*CALL SROTG (SA, SB, SC, SS)
*CALL DROTG (SA, SB, SC, SS)
```

Given the values *a* and *b*, these subprograms compute

$$c = \begin{cases} a/r & \text{if } r \neq 0 \\ 1 & \text{if } r = 0 \end{cases}$$

and

$$s = \begin{cases} b/r & \text{if } r \neq 0 \\ 1 & \text{if } r = 0 \end{cases}$$

where $r = \sigma(a^2 + b^2)^{1/2}$ and

$$\sigma = \begin{cases} \operatorname{sign}(a) & \text{if } |a| > |b| \\ \operatorname{sign}(b) & \text{otherwise} \end{cases}$$

Then, the values *c*, *s* and *r* satisfy the matrix equation

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

The introduction of σ is not essential to the computation of the Givens rotation matrix; but its use permits later stable reconstruction of *c* and *s* from just one stored number, an idea due to Stewart (1976). For this purpose, the subprogram also computes

$$z = \begin{cases} s & \text{if } |s| < c \text{ or } c = 0 \\ 1/c & \text{if } 0 < |c| \ \le \ s \end{cases}$$

In addition to returning *c* and *s*, the subprograms return *r* overwriting *a*, and *z* overwriting *b*.

Reconstruction of *c* and *s* from *z* can be done as follows:

If $z = 1$, then set $c = 0$ and $s = 1$

If $|z| < 1$, then set

$$c = \sqrt{1 - z^2} \quad \text{and} \quad s = z$$

If $|z| > 1$, then set

$$c = 1/z \quad \text{and} \quad s = \sqrt{1\text{-}c^2}$$

Apply a Plane Rotation

```
*CALL SROT (N, SX, INCX, SY, INCY, SC, SS)
*CALL DROT (N, DX, INCX, DY, INCY, DC, DS)
 CALL CSROT (N, CX, INCX, CY, INCY, SC, SS)
 CALL ZDROT (N, ZX, INCX, ZY, INCY, DC, DS)
```

These subprograms compute

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} x_i \\ y_i \end{bmatrix} \text{ for } i = 1, \ldots, N$$

If $N \le 0$, then the subprograms return immediately. CAUTION: For CSROT and ZDROT, the scalar quantities c and s are real, and x and y are complex.

Construct a Modified Givens Transformation

```
*CALL SROTMG (SD1, SD2, SX1, SY1, SPARAM)
*CALL DROTMG (DD1, DD2, DX1, DY1, DPARAM)
```

The input quantities d_1, d_2, x_1 and y_1 define a 2-vector $[w_1, z_1]^T$ by the following:

$$\begin{bmatrix} w_i \\ z_i \end{bmatrix} = \begin{bmatrix} \sqrt{d_1} & 0 \\ 0 & \sqrt{d_2} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

The subprograms determine the modified Givens rotation matrix H that transforms y_1, and thus, z_1 to zero. They also replace d_1, d_2 and x_1 with

$$\tilde{d}_1,\ \tilde{d}_2 \text{ and } \tilde{x}_1$$

respectively. That is,

$$\begin{bmatrix} \tilde{w}_1 \\ 0 \end{bmatrix} = \begin{bmatrix} \sqrt{\tilde{d}_1} & 0 \\ 0 & \sqrt{\tilde{d}_2} \end{bmatrix} H \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} \sqrt{\tilde{d}_1} & 0 \\ 0 & \sqrt{\tilde{d}_2} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} \tilde{x}_1 \\ 0 \end{bmatrix}$$

A representation of this matrix is stored in the array SPARAM or DPARAM. The form of the matrix H is flagged by PARAM(1).

PARAM(1) = 1. In this case,

$$\left|d_1 x_1^2\right| \le \left|d_2 y_1^2\right|$$

and

$$H = \begin{bmatrix} \text{PARAM}(2) & 1 \\ -1 & \text{PARAM}(5) \end{bmatrix}$$

The elements PARAM(3) and PARAM(4) are not changed.

PARAM(1) = 0. In this case,

$$\left|d_1 x_1^2\right| > \left|d_2 y_1^2\right|$$

and

$$H = \begin{bmatrix} 1 & \text{PARAM}(4) \\ \text{PARAM}(3) & 1 \end{bmatrix}$$

The elements PARAM(2) and PARAM(5) are not changed.

PARAM(1) = −1. In this case, rescaling was done and

$$H = \begin{bmatrix} \text{PARAM(2)} & \text{PARAM(4)} \\ \text{PARAM(3)} & \text{PARAM(5)} \end{bmatrix}$$

`PARAM`(1) = –2. In this case, $H = I$ where I is the identity matrix. The elements `PARAM`(2), `PARAM`(3), `PARAM`(4) and `PARAM`(5) are not changed.

The values of d_1, d_2 and x_1 are changed to represent the effect of the transformation. The quantity y_1, which would be zeroed by the transformation, is left unchanged.

The input value of d_1 should be nonnegative, but d_2 can be negative for the purpose of removing data from a least-squares problem.

See Lawson et al. (1979) for further details.

Apply a Modified Givens Transformation

```
*CALL SROTM (N, SX, INCX, SY, INCY, SPARAM)
*CALL DROTM (N, DX, INCX, DY, INCY, DPARAM)
 CALL CSROTM (N, CX, INCX, CY, INCY, SPARAM)
 CALL ZDROTM (N, ZX, INCX, ZY, INCY, DPARAM)
```

If `PARAM`(1) = 1.0, then these subprograms compute

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} \text{PARAM(2)} & 1 \\ -1 & \text{PARAM(5)} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \text{ for } i = 1, \ldots, N$$

If `PARAM`(1) = 0.0, then the subprograms compute

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} 1 & \text{PARAM(4)} \\ \text{PARAM(3)} & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \text{ for } i = 1, \ldots, N$$

If `PARAM`(1) = –1.0, then the subprograms compute

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} \text{PARAM(2)} & \text{PARAM(4)} \\ \text{PARAM(3)} & \text{PARAM(5)} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \text{ for } i = 1, \ldots, N$$

If $N \le 0$ or if `PARAM`(1) = –2.0, then the subprograms return immediately. CAUTION: For `CSROTM` and `ZDROTM`, the scalar quantities `PARAM`(`*`) are real and x and y are complex.

Programming Notes for Level 2 and Level 3 BLAS

For definitions of the matrix data structures used in the discussion below, see Reference Material. The Level 2 and Level 3 BLAS, like the Level 1 BLAS, do not follow the IMSL naming conventions. Instead, the names consist of a prefix of one of the letters “`S`,” “`D`,” “`C`” or “`Z`.” Next is a root name denoting the kind of matrix. This is followed by a suffix indicating the type of the operation.[1] The prefix denotes the type of operation according to the following table:

`S`	Real	`C`	Complex	
`D`	Double	`Z`	Double	Complex

The root names for the kind of matrix:

GE	General	GB	General Band
SY	Symmetric	SB	Symmetric Band
HE	Hermitian	HB	Hermitian Band
TR	Triangular	TB	Triangular Band

The suffixes for the type of operation:

MV	Matrix-Vector Product	SV	Solve for Vector
R	Rank-One Update		
RU	Rank-One Update, Unconjugated	RC	Rank-One Update, Conjugated
R2	Rank-Two Update		
MM	Matrix-Multiply	SM	Symmetric Matrix Multiply
RK	Rank-K Update	R2K	Rank 2K Update

[1] IMSL does not support the *Packed Symmetric, Packed-Hermitian,* or *Packed-Triangular* data structures, with respective root names SP, HP or TP, nor any extended precision versions of the Level 2 BLAS.

The specifications of the operations are provided by subprogram arguments of CHARACTER*1 data type. Both lower and upper case of the letter have the same meaning:

TRANS, TRANSA, TRANSB	'N'	No Transpose
	'T'	Transpose
	'C'	Conjugage and Transpose
UPLO	'L'	Lower Triangular
	'U'	Upper Triangular
DIAGNL	'N'	Non-unit Triangular
	'U'	Unit Triangular
SIDE	'L'	Multiply "*A*" Matrix on Left side or
	'R'	Right side of the "*B*" matrix

Note: See the "Triangular Mode" section in the Reference Material for definitions of these terms.

Descriptions of the Level 2 and Level 3 BLAS

The subprograms for Level 2 and Level 3 BLAS that perform operations involving the expression βy or βC do not require that the contents of y or C be defined when $\beta = 0$. In that case, the expression βy or βC is defined to be zero. Note that for the _GEMV and _GBMV subprograms, the dimensions of the vectors x and y are implied by the specification of the operation. If TRANS = 'N' , the dimension of y is m; if TRANS = 'T' or = 'C', the dimension of y is n. The Level 2 and Level 3 BLAS are summarized in Table 9.2. This table also lists the page numbers where the subprograms are described in more detail.

Specification of the Level 2 BLAS

Type and dimension for variables occurring in the subprogram specifications are

```
INTEGER           INCX, INCY, NCODA, NLCA, NUCA, LDA, M, N
CHARACTER*1       DIAGNL, TRANS, UPLO

REAL              SALPHA, SBETA, SX(*), SY(*), SA(LDA,*)
DOUBLE PRECISION  DALPHA, DBETA, DX(*), DY(*), DA(LDA,*)
COMPLEX           CALPHA, CBETA, CX(*), CY(*), CA(LDA,*)
DOUBLE COMPLEX    ZALPHA, ZBETA, ZX(*), ZY(*), ZA(LDA,*)
```

There is a lower bound on the leading dimension `LDA`. It must be ≥ the number of rows in the matrix that is contained in this array. Vector arguments have an increment parameter that specifies the storage space or stride between elements. The correspondence between the vector *x*, *y* and the arguments `SX`, `SY` and `INCX`, `INCY` is

$$x_i = \begin{cases} \mathrm{SX}\left(\left(\mathrm{I\text{-}1}\right) * \mathrm{INCX} + 1\right) & \text{if INCX } > 0 \\ \mathrm{SX}\left(\left(\mathrm{I\text{-}N}\right) * \mathrm{INCX} + 1\right) & \text{if INCX } < 0 \end{cases}$$

$$y_i = \begin{cases} \mathrm{SY}\left(\left(\mathrm{I\text{-}1}\right) * \mathrm{INCY} + 1\right) & \text{if INCY } > 0 \\ \mathrm{SY}\left(\left(\mathrm{I\text{-}N}\right) * \mathrm{INCY} + 1\right) & \text{if INCY } < 0 \end{cases}$$

In the Level 2 BLAS, only nonzero values of `INCX`, `INCY` are allowed for operations that have vector arguments. The Level 3 BLAS do not refer to `INCX`, `INCY`.

Specification of the Level 3 BLAS

Type and dimension for variables occurring in the subprogram specifications are

```
INTEGER           K, LDA, LDB, LDC, M, N
CHARACTER*1       DIAGNL, TRANS, TRANSA, TRANSB, SIDE, UPLO
REAL              SALPHA, SBETA, SA(LDA,*), SB(LDB,*),
                  SC(LDC,*)
DOUBLE PRECISION  DALPHA, DBETA, DA(LDA,*), DB(LDB,*),
                  DC(LDC,*)
COMPLEX           CALPHA, CBETA, CA(LDA,*), CB(LDB,*),
                  CC(LDC,*)
DOUBLE COMPLEX    ZALPHA, ZBETA, ZA(LDA,*), ZB(LDB,*),
                  ZC(LDC,*)
```

Each of the integers `K`, `M`, `N` must be ≥ 0. It is an error if any of them are < 0. If any of them are = 0, the subprograms return immediately. There are lower bounds on the leading dimensions `LDA`, `LDB`, `LDC`. Each must be ≥ the number of rows in the matrix that is contained in this array.

Table 9.2: Level 2 and 3 Basic Linear Algebra Subprograms

Operation	Real	Double	Complex	Double Complex	Pg.
Matrix-Vector Multiply, General	SGEMV	DGEMV	CGEMV	ZGEMV	1381
Matrix-Vector Multiply, Banded	SGBMV	DGBMV	CGBMV	ZGBMV	1381
Matrix-Vector Multiply, Hermitian			CHEMV	ZHEMV	1381

Operation	Real	Double	Complex	Double Complex	Pg.
Matrix-Vector Multiply, Hermitian and Banded			CHBMV	ZHBMV	1381
Matrix-Vector Multiply Symmetric and Real	SSYMV	DSYMV			1382
Matrix-Vector Multiply, Symmetric and Banded	SSBMV	DSBMV			1382
Matrix-Vector Multiply, Triangular	STRMV	DTRMV	CTRMV	ZTRMV	1382
Matrix-Vector Multiply, Triangular and Banded	STBMV	DTBMV	CTBMV	ZTBMV	1382
Matrix-Vector Solve, Triangular	STRSV	DTRSV	CTRSV	ZTRSV	1383
Matrix-Vector Solve, Triangular and Banded	STBSV	DTBSV	CTBSV	ZTBSV	1383
Rank-One Matrix Update, General and Real	SGER	DGER			1383
Rank-One Matrix Update, General, Complex and Transpose			CGERU	ZGERU	1384
Rank-One Matrix Update, General, Complex, and Conjugate Transpose			CGERC	ZGERC	1384
Rank-One Matrix Update, Hermitian and Conjugate Transpose			CHER	ZHER	1384
Rank-Two Matrix Update, Hermitian and Conjugate Transpose			CHER2	ZHER2	1384
Rank-One Matrix Update, Symmetric and Real	SSYR	DSYR			1384

Operation	Real	Double	Complex	Double Complex	Pg.
Rank-Two Matrix Update, Symmetric and Real	SSYR2	DSYR2			1384
Matrix--Matrix Multiply, General	SGEMM	DGEMM	CGEMM	ZGEMM	1385
Matrix-Matrix Multiply, Symmetric	SSYMM	DSYMM	CSYMM	ZSYMM	1385
Matrix-Matrix Multiply, Hermitian			CHEMM	ZHEMM	1385
Rank - *k* Update, Symmetric	SSYRK	DSYRK	CSYRK	ZSYRK	1386
Rank - *k* Update, Hermitian			CHERK	ZHERK	1386
Rank - 2*k* Update, Symmetric	SSYR2K	DSYR2K	CSYR2K	ZSYR2K	1386
Rank - 2*k* Update, Hermitian			CHER2K	ZHER2K	1386
Matrix-Matrix Multiply, Triangular	STRMM	DTRMM	CTRMM	ZTRMM	1387
Matrix-Matrix solve, Triangular	STRSM	DTRSM	CTRSM	ZTRSM	1387

Matrix–Vector Multiply, General

```
CALL SGEMV (TRANS, M, N, SALPHA, SA, LDA, SX, INCX,
            SBETA,SY, INCY)
CALL DGEMV (TRANS, M, N, DALPHA, DA, LDA, DX, INCX, DBETA,
            DY, INCY)
CALL CGEMV (TRANS, M, N, CALPHA, CA, LDA, CX, INCX, CBETA,
            CY, INCY)
CALL ZGEMV (TRANS, M, N, ZALPHA, ZA, LDA, ZX, INCX, ZBETA,
            ZY, INCY)
```

For all data types, A is an $M \times N$ matrix. These subprograms set y to one of the expressions: $y \leftarrow \alpha Ax + \beta y$, $y \leftarrow \alpha A^T x + \beta y$, or for complex data,

$$y \leftarrow \alpha \overline{A}^T + \beta y$$

The character flag `TRANS` determines the operation.

Matrix–Vector Multiply, Banded

```
CALL SGBMV (TRANS, M, N, NLCA, NUCA SALPHA, SA, LDA, SX,
            INCX, SBETA,SY, INCY)
CALL DGBMV (TRANS, M, N, NLCA, NUCA DALPHA, DA, LDA, DX,
            INCX, DBETA,DY, INCY)
CALL CGBMV (TRANS, M, N, NLCA, NUCA CALPHA, CA, LDA, CX,
            INCX, CBETA,CY, INCY)
CALL ZGBMV (TRANS, M, N, NLCA, NUCA ZALPHA, ZA, LDA, ZX,
            INCX, ZBETA,ZY, INCY)
```

For all data types, A is an $M \times N$ matrix with `NLCA` lower codiagonals and `NUCA` upper codiagonals. The matrix is stored in band storage mode. These subprograms set y to one of the expressions: $y \leftarrow \alpha Ax + \beta y$, $y \leftarrow \alpha A^T x + \beta y$, or for complex data,

$$y \leftarrow \alpha \overline{A}^T x + \beta y$$

The character flag `TRANS` determines the operation.

Matrix-Vector Multiply, Hermitian

```
CALL CHEMV (UPLO, N, CALPHA, CA, LDA, CX, INCX, CBETA,
            CY,INCY)
CALL ZHEMV (UPLO, N, ZALPHA, ZA, LDA, ZX, INCX, ZBETA, ZY,
            INCY)
```

For all data types, A is an $N \times N$ matrix. These subprograms set $y \leftarrow \alpha Ax + \beta y$ where A is an Hermitian matrix. The matrix A is either referenced using its upper or lower triangular part. The character flag `UPLO` determines the part used.

Matrix-Vector Multiply, Hermitian and Banded

```
CALL CHBMV (UPLO, N, NCODA, CALPHA, CA, LDA, CX, INCX,
            CBETA, CY,INCY)
CALL ZHBMV (UPLO, N, NCODA, ZALPHA, ZA, LDA, ZX, INCX,
            ZBETA, ZY,INCY)
```

For all data types, A is an $N \times N$ matrix with `NCODA` codiagonals. The matrix is stored in band Hermitian storage mode. These subprograms set $y \leftarrow \alpha Ax + \beta y$. The matrix A is either referenced using its upper or lower triangular part. The character flag `UPLO` determines the part used.

Matrix-Vector Multiply, Symmetric and Real

```
CALL SSYMV (UPLO, N, SALPHA, SA, LDA, SX, INCX, SBETA, SY,
            INCY)
CALL DSYMV (UPLO, N, DALPHA, DA, LDA, DX, INCX, DBETA, DY,
            INCY)
```

For all data types, A is an $N \times N$ matrix. These subprograms set $y \leftarrow \alpha Ax + \beta y$ where A is a symmetric matrix. The matrix A is either referenced using its upper or lower triangular part. The character flag `UPLO` determines the part used.

Matrix-Vector Multiply, Symmetric and Banded

```
CALL SSBMV (UPLO, N, NCODA, SALPHA, SA, LDA, SX, INCX,
            SBETA, SY,INCY)
CALL DSBMV (UPLO, N, NCODA, DALPHA, DA, LDA, DX, INCX,
            DBETA, DY,INCY)
```

For all data types, A is an $N \times N$ matrix with `NCODA` codiagonals. The matrix is stored in band symmetric storage mode. These subprograms set $y \leftarrow \alpha Ax + \beta y$. The matrix A is either referenced using its upper or lower triangular part. The character flag `UPLO` determines the part used.

Matrix-Vector Multiply, Triangular

```
CALL STRMV (UPLO, TRANS, DIAGNL, N, SA, LDA, SX, INCX)
CALL DTRMV (UPLO, TRANS, DIAGNL, N, DA, LDA, DX, INCX)
CALL CTRMV (UPLO, TRANS, DIAGNL, N, CA, LDA, CX, INCX)
CALL ZTRMV (UPLO, TRANS, DIAGNL, N, ZA, LDA, ZX, INCX)
```

For all data types, A is an $N \times N$ triangular matrix. These subprograms set x to one of the expressions: $x \leftarrow Ax$, $x \leftarrow A^T x$, or for complex data,

$$x \leftarrow \overline{A}^T x$$

The matrix A is either referenced using its upper or lower triangular part and is unit or nonunit triangular. The character flags `UPLO`, `TRANS`, and `DIAGNL` determine the part of the matrix used and the operation performed.

Matrix-Vector Multiply, Triangular and Banded

```
CALL STBMV (UPLO, TRANS, DIAGNL, N, NCODA, SA, LDA, SX,
INCX)
CALL DTBMV (UPLO, TRANS, DIAGNL, N, NCODA, DA, LDA, DX,
INCX)
CALL CTBMV (UPLO, TRANS, DIAGNL, N, NCODA, CA, LDA, CX,
INCX)
CALL ZTBMV (UPLO, TRANS, DIAGNL, N, NCODA, ZA, LDA, ZX,
INCX)
```

For all data types, A is an $N \times N$ matrix with `NCODA` codiagonals. The matrix is stored in band triangular storage mode. These subprograms set x to one of the expressions: $x \leftarrow Ax$, $x \leftarrow A^T x$, or for complex data,

$$x \leftarrow \overline{A}^T x$$

The matrix A is either referenced using its upper or lower triangular part and is unit or nonunit triangular. The character flags `UPLO`, `TRANS`, and `DIAGNL` determine the part of the matrix used and the operation performed.

Matrix-Vector Solve, Triangular

```
CALL STRSV (UPLO, TRANS, DIAGNL, N, SA, LDA, SX, INCX)
CALL DTRSV (UPLO, TRANS, DIAGNL, N, DA, LDA, DX, INCX)
CALL CTRSV (UPLO, TRANS, DIAGNL, N, CA, LDA, CX, INCX)
CALL ZTRSV (UPLO, TRANS, DIAGNL, N, ZA, LDA, ZX, INCX)
```

For all data types, A is an $N \times N$ triangular matrix. These subprograms solve x for one of the expressions: $x \leftarrow A^{-1} x$, $x \leftarrow (A^{-1})^T x$, or for complex data,

$$x \leftarrow \left(\overline{A}^T\right)^{-1} x$$

The matrix A is either referenced using its upper or lower triangular part and is unit or nonunit triangular. The character flags `UPLO`, `TRANS`, and `DIAGNL` determine the part of the matrix used and the operation performed.

Matrix-Vector Solve, Triangular and Banded

```
CALL STBSV (UPLO, TRANS, DIAGNL, N, NCODA, SA, LDA, SX,
INCX)
CALL DTBSV (UPLO, TRANS, DIAGNL, N, NCODA, DA, LDA, DX,
INCX)
CALL CTBSV (UPLO, TRANS, DIAGNL, N, NCODA, CA, LDA, CX,
INCX)
CALL ZTBSV (UPLO, TRANS, DIAGNL, N, NCODA, ZA, LDA, ZX,
INCX)
```

For all data types, A is an $N \times N$ triangular matrix with `NCODA` codiagonals. The matrix is stored in band triangular storage mode. These subprograms solve x for one of the expressions: $x \leftarrow A^{-1} x$, $x \leftarrow (A^T)^{-1} x$, or for complex data,

$$x \leftarrow \left(\overline{A}^T\right)^{-1} x$$

The matrix A is either referenced using its upper or lower triangular part and is unit or nonunit triangular. The character flags `UPLO`, `TRANS`, and `DIAGNL` determine the part of the matrix used and the operation performed.

Rank-One Matrix Update, General and Real

```
CALL SGER (M, N, SALPHA, SX, INCX, SY, INCY, SA, LDA)
CALL DGER (M, N, DALPHA, DX, INCX, DY, INCY, DA, LDA
```

For all data types, A is an $M \times N$ matrix. These subprograms set $A \leftarrow A + \alpha x y^T$.

Rank-One Matrix Update, General, Complex, and Transpose

```
CALL CGERU (M, N, CALPHA, CX, INCX, CY, INCY, CA, LDA)
CALL ZGERU (M, N, ZALPHA, ZX, INCX, ZY, INCY, ZA, LDA)
```

For all data types, A is an $M \times N$ matrix. These subprograms set $A \leftarrow A + \alpha xy^T$.

Rank-One Matrix Update, General, Complex, and Conjugate Transpose

```
CALL CGERC (M, N, CALPHA, CX, INCX, CY, INCY, CA, LDA)
CALL ZGERC (M, N, ZALPHA, ZX, INCX, ZY, INCY, ZA, LDA)
```

For all data types, A is an $M \times N$ matrix. These subprograms set

$$A \leftarrow A + \alpha x \overline{y}^T$$

Rank-One Matrix Update, Hermitian and Conjugate Transpose

```
CALL CHER (UPLO, N, SALPHA, CX, INCX, CA, LDA)
CALL ZHER (UPLO, N, DALPHA, ZX, INCX, ZA, LDA)
```

For all data types, A is an $N \times N$ matrix. These subprograms set

$$A \leftarrow A + \alpha x \overline{x}^T$$

where A is Hermitian. The matrix A is either referenced by its upper or lower triangular part. The character flag `UPLO` determines the part used. CAUTION: Notice the scalar parameter α is real, and the data in the matrix and vector are complex.

Rank-Two Matrix Update, Hermitian and Conjugate Transpose

```
CALL CHER2 (UPLO, N, CALPHA, CX, INCX, CY, INCY, CA, LDA)
CALL ZHER2 (UPLO, N, ZALPHA, ZX, INCX, ZY, INCY, ZA, LDA)
```

For all data types, A is an $N \times N$ matrix. These subprograms set

$$A \leftarrow A + \alpha x \overline{y}^T + \overline{\alpha}\, y \overline{x}^T$$

where A is an Hermitian matrix. The matrix A is either referenced by its upper or lower triangular part. The character flag `UPLO` determines the part used.

Rank-One Matrix Update, Symmetric and Real

```
CALL SSYR (UPLO, N, SALPHA, SX, INCX, SA, LDA)
CALL DSYR (UPLO, N, DALPHA, DX, INCX, DA, LDA)
```

For all data types, A is an $N \times N$ matrix. These subprograms set $A \leftarrow A + \alpha xx^T$ where A is a symmetric matrix. The matrix A is either referenced by its upper or lower triangular part. The character flag `UPLO` determines the part used.

Rank-Two Matrix Update, Symmetric and Real

```
CALL SSYR2 (UPLO, N, SALPHA, SX, INCX, SY, INCY, SA, LDA)
CALL DSYR2 (UPLO, N, DALPHA, DX, INCX, DY, INCY, DA, LDA)
```

For all data types, A is an $N \times N$ matrix. These subprograms set $A \leftarrow A + \alpha xy^T + \alpha yx^T$ where A is a symmetric matrix. The matrix A is referenced by its upper or lower triangular part. The character flag `UPLO` determines the part used.

Matrix-Matrix Multiply, General

```
CALL SGEMM (TRANSA, TRANSB, M, N, K, SALPHA, SA, LDA, SB,
            LDB, SBETA, SC, LDC)
CALL DGEMM (TRANSA, TRANSB, M, N, K, DALPHA, DA, LDA, DB,
            LDB, DBETA, DC, LDC)
CALL CGEMM (TRANSA, TRANSB, M, N, K, CALPHA, CA, LDA, CB,
            LDB, CBETA, CC, LDC)
CALL ZGEMM (TRANSA, TRANSB, M, N, K, ZALPHA, ZA, LDA, ZB,
            LDB, ZBETA, ZC, LDC)
```

For all data types, these subprograms set $C_{M \times N}$ to one of the expressions:

$$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha A^T B + \beta C, C \leftarrow \alpha AB^T + \beta C, C \leftarrow \alpha A^T B^T + \beta C,$$

or for complex data, $C \leftarrow \alpha A\overline{B}^T + \beta C, C \leftarrow \alpha \overline{A}^T B + \beta C, C \leftarrow \alpha A^T \overline{B}^T + \beta C,$

$$C \leftarrow \alpha \overline{A}^T B^T + \beta C, C \leftarrow \alpha \overline{A}^T \overline{B}^T + \beta C$$

The character flags `TRANSA` and `TRANSB` determine the operation to be performed. Each matrix product has dimensions that follow from the fact that C has dimension $M \times N$.

Matrix-Matrix Multiply, Symmetric

```
CALL SSYMM (SIDE, UPLO, M, N, SALPHA, SA, LDA, SB, LDB,
            SBETA, SC, LDC)
CALL DSYMM (SIDE, UPLO, M, N, DALPHA, DA, LDA, DB, LDB,
            DBETA, DC, LDC)
CALL CSYMM (SIDE, UPLO, M, N, CALPHA, CA, LDA, CB, LDB,
            CBETA, CC, LDC)
CALL ZSYMM (SIDE, UPLO, M, N, ZALPHA, ZA, LDA, ZB, LDB,
            ZBETA, ZC, LDC)
```

For all data types, these subprograms set $C_{M \times N}$ to one of the expressions: $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$, where A is a symmetric matrix. The matrix A is referenced either by its upper or lower triangular part. The character flags `SIDE` and `UPLO` determine the part of the matrix used and the operation performed.

Matrix-Matrix Multiply, Hermitian

```
CALL CHEMM (SIDE, UPLO, M, N, CALPHA, CA, LDA, CB, LDB,
            CBETA, CC, LDC)
CALL ZHEMM (SIDE, UPLO, M, N, ZALPHA, ZA, LDA, ZB, LDB,
            ZBETA, ZC, LDC)
```

For all data types, these subprograms set $C_{M \times N}$ to one of the expressions: $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$, where A is an Hermitian matrix. The matrix A is referenced either by its upper or lower triangular part. The character flags `SIDE` and `UPLO` determine the part of the matrix used and the operation performed.

Rank-*k* Update, Symmetric

```
CALL SSYRK (UPLO, TRANS, N, K, SALPHA, SA, LDA, SBETA, SC,
            LDC)
CALL DSYRK (UPLO, TRANS, N, K, DALPHA, DA, LDA, DBETA, DC,
            LDC)
CALL CSYRK (UPLO, TRANS, N, K, CALPHA, CA, LDA, CBETA, CC,
            LDC)
CALL ZSYRK (UPLO, TRANS, N, K, ZALPHA, ZA, LDA, ZBETA, ZC,
            LDC)
```

For all data types, these subprograms set $C_{M \times N}$ to one of the expressions: $C \leftarrow \alpha AA^T + \beta C$ or $C \leftarrow \alpha A^T A + \beta C$. The matrix C is referenced either by its upper or lower triangular part. The character flags `UPLO` and `TRANS` determine the part of the matrix used and the operation performed. In subprogram `CSYRK` and `ZSYRK`, only values `'N'` or `'T'` are allowed for `TRANS`; `'C'` is not acceptable.

Rank-*k* Update, Hermitian

```
CALL CHERK (UPLO, TRANS, N, K, SALPHA, CA, LDA, SBETA, CC,
            LDC)
CALL ZHERK (UPLO, TRANS, N, K, DALPHA, ZA, LDA, DBETA, ZC,
            LDC)
```

For all data types, these subprograms set $C_{N \times N}$ to one of the expressions:

$$C \leftarrow \alpha A\overline{A}^T + \beta C \text{ or } C \leftarrow \alpha \overline{A}^T A + \beta C$$

The matrix C is referenced either by its upper or lower triangular part. The character flags `UPLO` and `TRANS` determine the part of the matrix used and the operation performed. CAUTION: Notice the scalar parameters α and β are real, and the data in the matrices are complex. Only values `'N'` or `'C'` are allowed for `TRANS`; `'T'` is not acceptable.

Rank-2*k* Update, Symmetric

```
CALL SSYR2K (UPLO, TRANS, N, K, SALPHA, SA, LDA, SB, LDB,
             SBETA, SC, LDC)
CALL DSYR2K (UPLO, TRANS, N, K, DALPHA, DA, LDA, DB, LDB,
             DBETA, DC, LDC)
CALL CSYR2K (UPLO, TRANS, N, K, CALPHA, CA, LDA, CB, LDB,
             CBETA, CC, LDC)
CALL ZSYR2K (UPLO, TRANS, N, K, ZALPHA, ZA, LDA, ZB, LDB,
             ZBETA, ZC, LDC)
```

For all data types, these subprograms set $C_{N \times N}$ to one of the expressions:

$$C \leftarrow \alpha AB^T + \alpha\beta A^T + \beta \mathrm{C} \text{ or } C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$$

The matrix C is referenced either by its upper or lower triangular part. The character flags `UPLO` and `TRANS` determine the part of the matrix used and the operation performed. In subprogram `CSYR2K` and `ZSYR2K`, only values `'N'` or `'T'` are allowed for `TRANS`; `'C'` is not acceptable.

Rank-2*k* Update, Hermitian

```
CALL CHER2K (UPLO, TRANS, N, K, CALPHA, CA, LDA, CB, LDB,
             SBETA, CC, LDC)
CALL ZHER2K (UPLO, TRANS, N, K, ZALPHA, ZA, LDA, ZB, LDB,
             DBETA, ZC, LDC)
```

For all data types, these subprograms set $C_{N \times N}$ to one of the expressions:

$$C \leftarrow \alpha A\overline{B}^T + \overline{\alpha} B\overline{A}^T + \beta \mathrm{C} \text{ or } C \leftarrow \alpha \overline{A}^T B + \overline{\alpha}\,\overline{B}^T A + \beta C$$

The matrix *C* is referenced either by its upper or lower triangular part. The character flags `UPLO` and `TRANS` determine the part of the matrix used and the operation performed. CAUTION: Notice the scalar parameter β is real, and the data in the matrices are complex. In subprogram `CHER2K` and `ZHER2K`, only values `'N'` or `'C'` are allowed for `TRANS`; `'T'` is not acceptable.

Matrix-Matrix Multiply, Triangular

```
CALL STRMM (SIDE, UPLO, TRANSA, DIAGNL, M, N, SALPHA, SA,
            LDA, SB, LDB)
CALL DTRMM (SIDE, UPLO, TRANSA, DIAGNL, M, N, DALPHA, DA,
            LDA, DB, LDB)
CALL CTRMM (SIDE, UPLO, TRANSA, DIAGNL, M, N, CALPHA, CA,
            LDA, CB,LDB)
CALL ZTRMM (SIDE, UPLO, TRANSA, DIAGNL, M, N, ZALPHA, ZA,
            LDA, ZB, LDB)
```

For all data types, these subprograms set $B_{M \times N}$ to one of the_expressions:

$$B \leftarrow \alpha AB,\ B \leftarrow \alpha A^T B,\ B \leftarrow \alpha BA,\ B \leftarrow \alpha BA^T,$$

$$\text{or for complex data, } B \leftarrow \alpha \overline{A}^T B, \text{ or } B \leftarrow \alpha B\overline{A}^T$$

where *A* is a triangular matrix. The matrix *A* is either referenced using its upper or lower triangular part and is unit or nonunit triangular. The character flags `SIDE`, `UPLO`, `TRANSA`, and `DIAGNL` determine the part of the matrix used and the operation performed.

Matrix-Matrix Solve, Triangular

```
CALL STRSM (SIDE, UPLO, TRANSA, DIAGNL, M, N, SALPHA, SA,
            LDA, SB, LDB)
CALL DTRSM (SIDE, UPLO, TRANSA, DIAGNL, M, N, DALPHA, DA,
            LDA, DB, LDB)
CALL CTRSM (SIDE, UPLO, TRANSA, DIAGNL, M, N, CALPHA, CA,
            LDA, CB, LDB)
CALL ZTRSM (SIDE, UPLO, TRANSA, DIAGNL, M, N, ZALPHA, ZA,
            LDA, ZB, LDB)
```

For all data types, these subprograms set $B_{M \times N}$ to one of the expressions:

$$B \leftarrow \alpha A^{-1}B,\ B \leftarrow \alpha BA^{-1},\ B \leftarrow \alpha \left(A^{-1}\right)^{\mathrm{T}} B,\ B \leftarrow \alpha B\left(A^{-1}\right)^T,$$

$$\text{or for complex data, } B \leftarrow \alpha \left(\overline{A}^T\right)^{-1} B, \text{ or } B \leftarrow \alpha B\left(\overline{A}^T\right)^{-1}$$

where A is a triangular matrix. The matrix A is either referenced using its upper or lower triangular part and is unit or nonunit triangular. The character flags SIDE, UPLO, TRANSA, and DIAGNL determine the part of the matrix used and the operation performed.

Other Matrix/Vector Operations

This section describes a set of routines for matrix/vector operations. The matrix copy and conversion routines are summarized by the following table:

From	**To**			
	Real General	**Complex General**	**Real Band**	**Complex Band**
Real General	CRGRG p. 1389	CRGCG p. 1402	CRGRB p. 1395	
Complex General		CCGCG p. 1390		CCGCB p. 1398
Real Band	CRBRG p. 1397		CRBRB p. 1392	CRBCB p. 1405
Complex Band		CCBCG p. 1400		CCBCB p. 1393
Symmetric Full	CSFRG p. 1406			
Hermitian Full		CHFCG p. 1408		
Symmetric Band			CSBRB p. 1409	
Hermitian Band				CHBCB p. 1411

The matrix multiplication routines are summarized as follows:

AB	***A***			
B	**Real Rect.**	**Complex Rect.**	**Real Band**	**Complex Band**
Real Rectangular	MRRRR p. 1421			
Complex Rect.		MCRCR p. 1423		
Vector	MURRV p. 1431	MUCRV p. 1435	MURBV p. 1433	MUCBV p. 1436

The matrix norm routines are summarized as follows:

||*A*||	Real Rectangular	Real Band	Complex Band
∞-norm	NRIRR p. 1443		
1-norm	NR1RR p. 1444	NR1RB p. 1447	NR1CB p. 1449
Frobenius	NR2RR p. 1446		

CRGRG

Copies a real general matrix.

Required Arguments

A — Matrix of order `N`. (Input)

B — Matrix of order `N` containing a copy of `A`. (Output)

Optional Arguments

N — Order of the matrices. (Input)
Default: `N` = size (`A`,2).

LDA — Leading dimension of `A` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDA` = size (`A`,1).

LDB — Leading dimension of `B` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDB` = size (`B`,1).

FORTRAN 90 Interface

Generic: `CALL CRGRG (A, B [,…])`

Specific: The specific interface names are `S_CRGRG` and `D_CRGRG`.

FORTRAN 77 Interface

Single: `CALL CRGRG (N, A, LDA, B, LDB)`

Double: The double precision name is `DCRGRG`.

Example

A real 3 × 3 general matrix is copied into another real 3 × 3 general matrix.

```
      USE CRGRG_INT
      USE WRRRN_INT
!                                 Declare variables
      INTEGER    LDA, LDB, N
      PARAMETER  (LDA=3, LDB=3, N=3)
!
      REAL       A(LDA,N), B(LDB,N)
!                                 Set values for  A
!                                 A = (   0.0   1.0   1.0  )
!                                     (  -1.0   0.0   1.0  )
!                                     (  -1.0  -1.0   0.0  )
!
      DATA A/0.0, 2* - 1.0, 1.0, 0.0, -1.0, 2*1.0, 0.0/
!                                 Copy real matrix A to real matrix B
      CALL CRGRG (A, B)
!                                 Print results
      CALL WRRRN ('B', B)
      END
```

Output

```
            B
        1       2       3
1   0.000   1.000   1.000
2  -1.000   0.000   1.000
3  -1.000  -1.000   0.000
```

Description

The routine CRGRG copies the real $N \times N$ general matrix A into the real $N \times N$ general matrix B.

CCGCG

Copies a complex general matrix.

Required Arguments

A — Complex matrix of order N. (Input)

B — Complex matrix of order N containing a copy of A. (Output)

Optional Arguments

N — Order of the matrices A and B. (Input)
Default: N = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

FORTRAN 90 Interface

Generic: CALL CCGCG (A, B [,…])

Specific: The specific interface names are S_CCGCG and D_CCGCG.

FORTRAN 77 Interface

Single: CALL CCGCG (N, A, LDA, B, LDB)

Double: The double precision name is DCCGCG.

Example

A complex 3 × 3 general matrix is copied into another complex 3 × 3 general matrix.

```
      USE CCGCG_INT
      USE WRCRN_INT
!                                 Declare variables
      INTEGER    LDA, LDB, N
      PARAMETER  (LDA=3, LDB=3, N=3)
!
      COMPLEX    A(LDA,N), B(LDB,N)
!                          Set values for  A
!                          A = (  0.0+0.0i  1.0+1.0i  1.0+1.0i  )
!                              ( -1.0-1.0i  0.0+0.0i  1.0+1.0i  )
!                              ( -1.0-1.0i -1.0-1.0i  0.0+0.0i  )
!
      DATA A/(0.0,0.0), 2*(-1.0,-1.0), (1.0,1.0), (0.0,0.0), &
          (-1.0,-1.0), 2*(1.0,1.0), (0.0,0.0)/
!                                 Copy matrix A to matrix B
      CALL CCGCG (A, B)
!                                 Print results
      CALL WRCRN ('B', B)
      END
```

Output

```
                          B
              1                  2                  3
1  ( 0.000, 0.000)  ( 1.000, 1.000)  ( 1.000, 1.000)
2  (-1.000,-1.000)  ( 0.000, 0.000)  ( 1.000, 1.000)
3  (-1.000,-1.000)  (-1.000,-1.000)  ( 0.000, 0.000)
```

Description

The routine CCGCG copies the complex $N \times N$ general matrix A into the complex $N \times N$ general matrix B.

CRBRB

Copies a real band matrix stored in band storage mode.

Required Arguments

A — Real band matrix of order N. (Input)

NLCA — Number of lower codiagonals in A. (Input)

NUCA — Number of upper codiagonals in A. (Input)

B — Real band matrix of order N containing a copy of A. (Output)

NLCB — Number of lower codiagonals in B. (Input)
NLCB must be at least as large as NLCA.

NUCB — Number of upper codiagonals in B. (Input)
NUCB must be at least as large as NUCA.

Optional Arguments

N — Order of the matrices A and B. (Input)
Default: N = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

FORTRAN 90 Interface

Generic: `CALL CRBRB (A, NLCA, NUCA, B, NLCB, NUCB [,…])`

Specific: The specific interface names are S_CRBRB and D_CRBRB.

FORTRAN 77 Interface

Single: `CALL CRBRB (N, A, LDA, NLCA, NUCA, B, LDB, NLCB, NUCB)`

Double: The double precision name is DCRBRB.

Example

A real band matrix of order 3, in band storage mode with one upper codiagonal, and one lower codiagonal is copied into another real band matrix also in band storage mode.

```
      USE CRBRB_INT
      USE WRRRN_INT
!                                 Declare variables
      INTEGER    LDA, LDB, N, NLCA, NLCB, NUCA, NUCB
      PARAMETER  (LDA=3, LDB=3, N=3, NLCA=1, NLCB=1, NUCA=1, NUCB=1)
!
      REAL       A(LDA,N), B(LDB,N)
!                                 Set values for  A (in band mode)
!                                 A = (  0.0  1.0   1.0  )
!                                     (  1.0  1.0   1.0  )
!                                     (  1.0  1.0   0.0  )
!
      DATA A/0.0, 7*1.0, 0.0/
!                                 Copy A to B
      CALL CRBRB (A, NLCA, NUCA, B, NLCB, NUCB)
!                                 Print results
      CALL WRRRN (’B’, B)
      END
```

Output

```
             B
        1       2       3
1   0.000   1.000   1.000
2   1.000   1.000   1.000
3   1.000   1.000   0.000
```

Description

The routine CRBRB copies the real band matrix *A* in band storage mode into the real band matrix *B* in band storage mode.

CCBCB

Copies a complex band matrix stored in complex band storage mode.

Required Arguments

A — Complex band matrix of order N. (Input)

NLCA — Number of lower codiagonals in A. (Input)

NUCA — Number of upper codiagonals in A. (Input)

B — Complex matrix of order N containing a copy of A. (Output)

NLCB — Number of lower codiagonals in B. (Input)
NLCB must be at least as large as NLCA.

NUCB — Number of upper codiagonals in B. (Input)
NUCB must be at least as large as NUCA.

Optional Arguments

N — Order of the matrices A and B. (Input)
Default: N = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

FORTRAN 90 Interface

Generic: CALL CCBCB (A, NLCA, NUCA, B, NLCB, NUCB [,…])

Specific: The specific interface names are S_CCBCB and D_CCBCB.

FORTRAN 77 Interface

Single: CALL CCBCB (N, A, LDA, NLCA, NUCA, B, LDB, NLCB, NUCB)

Double: The double precision name is DCCBCB.

Example

A complex band matrix of order 3 in band storage mode with one upper codiagonal and one lower codiagonal is copied into another complex band matrix in band storage mode.

```
      USE CCBCB_INT
      USE WRCRN_INT
!                                 Declare variables
      INTEGER    LDA, LDB, N, NLCA, NLCB, NUCA, NUCB
      PARAMETER  (LDA=3, LDB=3, N=3, NLCA=1, NLCB=1, NUCA=1, NUCB=1)
!
      COMPLEX    A(LDA,N), B(LDB,N)
!                        Set values for  A (in band mode)
!                        A = (  0.0+0.0i  1.0+1.0i  1.0+1.0i  )
!                            (  1.0+1.0i  1.0+1.0i  1.0+1.0i  )
!                            (  1.0+1.0i  1.0+1.0i  0.0+0.0i  )
!
      DATA A/(0.0,0.0), 7*(1.0,1.0), (0.0,0.0)/
!                                 Copy A to B
```

```
      CALL CCBCB (A, NLCA, NUCA, B, NLCB, NUCB)
!                                Print results
      CALL WRCRN ('B', B)
      END
```

Output

```
                          B
              1                  2                  3
1  ( 0.000, 0.000)  ( 1.000, 1.000)  ( 1.000, 1.000)
2  ( 1.000, 1.000)  ( 1.000, 1.000)  ( 1.000, 1.000)
3  ( 1.000, 1.000)  ( 1.000, 1.000)  ( 0.000, 0.000)
```

Description

The routine CCBCB copies the complex band matrix *A* in band storage mode into the complex band matrix *B* in band storage mode.

CRGRB

Converts a real general matrix to a matrix in band storage mode.

Required Arguments

A — Real N by N matrix. (Input)

NLC — Number of lower codiagonals in B. (Input)

NUC — Number of upper codiagonals in B. (Input)

B — Real (NUC + 1 + NLC) by N array containing the band matrix in band storage mode. (Output)

Optional Arguments

N — Order of the matrices A and B. (Input)
Default: N = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

FORTRAN 90 Interface

Generic: CALL CRGRB (A, NLC, NUC, B [,…])

Specific: The specific interface names are S_CRGRB and D_CRGRB.

FORTRAN 77 Interface

Single: `CALL CRGRB (N, A, LDA, NLC, NUC, B, LDB)`

Double: The double precision name is DCRGRB.

Example

A real 4 × 4 matrix with one upper codiagonal and three lower codiagonals is copied to a real band matrix of order 4 in band storage mode.

```
      USE CRGRB_INT
      USE WRRRN_INT
!                                 Declare variables
      INTEGER    LDA, LDB, N, NLC, NUC
      PARAMETER  (LDA=4, LDB=5, N=4, NLC=3, NUC=1)
!
      REAL       A(LDA,N), B(LDB,N)
!                                 Set values for  A
!                                 A = (  1.0     2.0    0.0    0.0)
!                                     ( -2.0     1.0    3.0    0.0)
!                                     (  0.0    -3.0    1.0    4.0)
!                                     ( -7.0     0.0   -4.0    1.0)
!
      DATA A/1.0, -2.0, 0.0, -7.0, 2.0, 1.0, -3.0, 0.0, 0.0, 3.0, 1.0, &
          -4.0, 0.0, 0.0, 4.0, 1.0/
!                                 Convert A to band matrix B
      CALL CRGRB (A, NLC, NUC, B)
!                                 Print results
      CALL WRRRN ('B', B)
      END
```

Output

```
                 B
        1        2        3        4
1   0.000    2.000    3.000    4.000
2   1.000    1.000    1.000    1.000
3  -2.000   -3.000   -4.000    0.000
4   0.000    0.000    0.000    0.000
5  -7.000    0.000    0.000    0.000
```

Description

The routine CRGRB converts the real general $N \times N$ matrix A with m_u = NUC upper codiagonals and m_l = NLC lower codiagonals into the real band matrix B of order N. The first m_u rows of B then contain the upper codiagonals of A, the next row contains the main diagonal of A, and the last m_l rows of B contain the lower codiagonals of A.

CRBRG

Converts a real matrix in band storage mode to a real general matrix.

Required Arguments

A — Real (NUC + 1 + NLC) by N array containing the band matrix in band storage mode. (Input)

NLC — Number of lower codiagonals in A. (Input)

NUC — Number of upper codiagonals in A. (Input)

B — Real N by N array containing the matrix. (Output)

Optional Arguments

N — Order of the matrices A and B. (Input)
Default: N = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

FORTRAN 90 Interface

Generic: CALL CRBRG (A, NLC, NUC, B [,…])

Specific: The specific interface names are S_CRBRG and D_CRBRG.

FORTRAN 77 Interface

Single: CALL CRBRG (N, A, LDA, NLC, NUC, B, LDB)

Double: The double precision name is DCRBRG.

Example

A real band matrix of order 3 in band storage mode with one upper codiagonal and one lower codiagonal is copied to a 3 × 3 real general matrix.

```
      USE CRBRG_INT
      USE WRRRN_INT
!                                 Declare variables
```

```
      INTEGER    LDA, LDB, N, NLC, NUC
      PARAMETER  (LDA=3, LDB=3, N=3, NLC=1, NUC=1)
!
      REAL       A(LDA,N), B(LDB,N)
!                                 Set values for  A (in band mode)
!                                 A = (  0.0     1.0    1.0)
!                                     (  4.0     3.0    2.0)
!                                     (  2.0     2.0    0.0)
!
      DATA A/0.0, 4.0, 2.0, 1.0, 3.0, 2.0, 1.0, 2.0, 0.0/
!                                 Convert band matrix A to matrix B
      CALL CRBRG (A, NLC, NUC, B)
!                                 Print results
      CALL WRRRN ('B', B)
      END
```

Output

```
             B
        1       2       3
1   4.000   1.000   0.000
2   2.000   3.000   1.000
3   0.000   2.000   2.000
```

Description

The routine CRBRG converts the real band matrix *A* of order *N* in band storage mode into the real *N* × *N* general matrix *B* with m_u = NUC upper codiagonals and m_l = NLC lower codiagonals. The first m_u rows of *A* are copied to the upper codiagonals of *B*, the next row of *A* is copied to the diagonal of *B*, and the last m_l rows of *A* are copied to the lower codiagonals of *B*.

CCGCB

Converts a complex general matrix to a matrix in complex band storage mode.

Required Arguments

A — Complex N by N array containing the matrix. (Input)

NLC — Number of lower codiagonals in B. (Input)

NUC — Number of upper codiagonals in B. (Input)

B — Complex (NUC + 1 + NLC) by N array containing the band matrix in band storage mode. (Output)

Optional Arguments

N — Order of the matrices A and B. (Input)
Default: N = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

FORTRAN 90 Interface

Generic: CALL CCGCB (A, NLC, NUC, B [,…])

Specific: The specific interface names are S_CCGCB and D_CCGCB.

FORTRAN 77 Interface

Single: CALL CCGCB (N, A, LDA, NLC, NUC, B, LDB)

Double: The double precision name is DCCGCB.

Example

A complex general matrix of order 4 with one upper codiagonal and three lower codiagonals is copied to a complex band matrix of order 4 in band storage mode.

```
      USE CCGCB_INT
      USE WRCRN_INT
!                                 Declare variables
      INTEGER    LDA, LDB, N, NLC, NUC
      PARAMETER  (LDA=4, LDB=5, N=4, NLC=3, NUC=1)
!
      COMPLEX    A(LDA,N), B(LDB,N)
!                      Set values for  A
!                      A = (  1.0+0.0i   2.0+1.0i  0.0+0.0i  0.0+0.0i )
!                          ( -2.0+1.0i   1.0+0.0i  3.0+2.0i  0.0+0.0i )
!                          (  0.0+0.0i  -3.0+2.0i  1.0+0.0i  4.0+3.0i )
!                          ( -7.0+1.0i   0.0+0.0i -4.0+3.0i  1.0+0.0i )
!
      DATA A/(1.0,0.0), (-2.0,1.0), (0.0,0.0), (-7.0,1.0), (2.0,1.0), &
          (1.0,0.0), (-3.0,2.0), (0.0,0.0), (0.0,0.0), (3.0,2.0), &
          (1.0,0.0), (-4.0,3.0), (0.0,0.0), (0.0,0.0), (4.0,3.0), &
          (1.0,0.0)/
!                                 Convert A to band matrix B
      CALL CCGCB (A, NLC, NUC, B)
!                                 Print results
      CALL WRCRN ('B', B)
      END
```

Output

```
                                B
             1                 2                 3                 4
```

```
1  ( 0.000, 0.000)  ( 2.000, 1.000)  ( 3.000, 2.000)  ( 4.000, 3.000)
2  ( 1.000, 0.000)  ( 1.000, 0.000)  ( 1.000, 0.000)  ( 1.000, 0.000)
3  (-2.000, 1.000)  (-3.000, 2.000)  (-4.000, 3.000)  ( 0.000, 0.000)
4  ( 0.000, 0.000)  ( 0.000, 0.000)  ( 0.000, 0.000)  ( 0.000, 0.000)
5  (-7.000, 1.000)  ( 0.000, 0.000)  ( 0.000, 0.000)  ( 0.000, 0.000)
```

Description

The routine `CCGCB` converts the complex general matrix *A* of order *N* with m_u = `NUC` upper codiagonals and m_l = `NLC` lower codiagonals into the complex band matrix *B* of order *N* in band storage mode. The first m_u rows of *B* then contain the upper codiagonals of *A*, the next row contains the main diagonal of *A*, and the last m_l rows of *B* contain the lower codiagonals of *A*.

CCBCG

Converts a complex matrix in band storage mode to a complex matrix in full storage mode.

Required Arguments

A — Complex (`NUC` + 1 + `NLC`) by `N` matrix containing the band matrix in band mode. (Input)

NLC — Number of lower codiagonals in `A`. (Input)

NUC — Number of upper codiagonals in `A`. (Input)

B — Complex `N` by `N` matrix containing the band matrix in full mode. (Output)

Optional Arguments

N — Order of the matrices `A` and `B`. (Input)
Default: `N` = size (`A`,2).

LDA — Leading dimension of `A` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDA` = size (`A`,1).

LDB — Leading dimension of `B` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDB` = size (`B`,1).

FORTRAN 90 Interface

Generic: `CALL CCBCG (A, NLC, NUC, B [,…])`

Specific: The specific interface names are `S_CCBCG` and `D_CCBCG`.

FORTRAN 77 Interface

Single: `CALL CCBCG (N, A, LDA, NLC, NUC, B, LDB)`

Double: The double precision name is `DCCBCG`.

Example

A complex band matrix of order 4 in band storage mode with one upper codiagonal and three lower codiagonals is copied into a 4 × 4 complex general matrix.

```
      USE CCBCG_INT
      USE WRCRN_INT
!                                 Declare variables
      INTEGER    LDA, LDB, N, NLC, NUC
      PARAMETER  (LDA=5, LDB=4, N=4, NLC=3, NUC=1)
!
      COMPLEX    A(LDA,N), B(LDB,N)
!                   Set values for  A (in band mode)
!                   A = (  0.0+0.0i  2.0+1.0i  3.0+2.0i  4.0+3.0i  )
!                       (  1.0+0.0i  1.0+0.0i  1.0+0.0i  1.0+0.0i  )
!                       ( -2.0+1.0i -3.0+2.0i -4.0+3.0i  0.0+0.0i  )
!                       (  0.0+0.0i  0.0+0.0i  0.0+0.0i  0.0+0.0i  )
!                       ( -7.0+1.0i  0.0+0.0i  0.0+0.0i  0.0+0.0i  )
!
      DATA A/(0.0,0.0), (1.0,0.0), (-2.0,1.0), (0.0,0.0), (-7.0,1.0), &
          (2.0,1.0), (1.0,0.0), (-3.0,2.0), 2*(0.0,0.0), (3.0,2.0), &
          (1.0,0.0), (-4.0,3.0), 2*(0.0,0.0), (4.0,3.0), (1.0,0.0), &
          3*(0.0,0.0)/
!                                 Convert band matrix A to matrix B
      CALL CCBCG (A, NLC, NUC, B)
!                                 Print results
      CALL WRCRN ('B', B)
      END
```

Output

```
                                    B
                  1                  2                  3                  4
1  ( 1.000, 0.000)  ( 2.000, 1.000)  ( 0.000, 0.000)  ( 0.000, 0.000)
2  (-2.000, 1.000)  ( 1.000, 0.000)  ( 3.000, 2.000)  ( 0.000, 0.000)
3  ( 0.000, 0.000)  (-3.000, 2.000)  ( 1.000, 0.000)  ( 4.000, 3.000)
4  (-7.000, 1.000)  ( 0.000, 0.000)  (-4.000, 3.000)  ( 1.000, 0.000)
```

Description

The routine `CCBCG` converts the complex band matrix *A* of order *N* with m_u = `NUC` upper codiagonals and m_l = `NLC` lower codiagonals into the *N* × *N* complex general matrix *B*. The first m_u rows of *A* are copied to the upper codiagonals of *B*, the next row of *A* is copied to the diagonal of *B*, and the last m_l rows of *A* are copied to the lower codiagonals of *B*.

CRGCG

Copies a real general matrix to a complex general matrix.

Required Arguments

A — Real matrix of order N. (Input)

B — Complex matrix of order N containing a copy of A. (Output)

Optional Arguments

N — Order of the matrices A and B. (Input)
Default: N = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

FORTRAN 90 Interface

Generic: `CALL CRGCG (A, B [,…])`

Specific: The specific interface names are `S_CRGCG` and `D_CRGCG`.

FORTRAN 77 Interface

Single: `CALL CRGCG (N, A, LDA, B, LDB)`

Double: The double precision name is `DCRGCG`.

Example

A 3 × 3 real matrix is copied to a 3 × 3 complex matrix.

```
      USE CRGCG_INT
      USE WRCRN_INT
!                                 Declare variables
      INTEGER    LDA, LDB, N
      PARAMETER  (LDA=3, LDB=3, N=3)
!
      REAL       A(LDA,N)
      COMPLEX    B(LDB,N)
!                                 Set values for  A
!                                  A = (  2.0     1.0    3.0 )
!                                      (  4.0     1.0    0.0 )
!                                      ( -1.0     2.0    0.0 )
!
      DATA A/2.0, 4.0, -1.0, 1.0, 1.0, 2.0, 3.0, 0.0, 0.0/
!                                  Convert real A to complex B
      CALL CRGCG (A, B)
!                                  Print results
      CALL WRCRN ('B', B)
      END
```

Output

```
                            B
                 1                 2                 3
1  ( 2.000, 0.000)  ( 1.000, 0.000)  ( 3.000, 0.000)
2  ( 4.000, 0.000)  ( 1.000, 0.000)  ( 0.000, 0.000)
3  (-1.000, 0.000)  ( 2.000, 0.000)  ( 0.000, 0.000)
```

Description

The routine CRGCG copies a real $N \times N$ matrix to a complex $N \times N$ matrix.

CRRCR

Copies a real rectangular matrix to a complex rectangular matrix.

Required Arguments

A — Real NRA by NCA rectangular matrix. (Input)

B — Complex NRB by NCB rectangular matrix containing a copy of A. (Output)

Optional Arguments

NRA — Number of rows in A. (Input)
Default: NRA = size (A,1).

NCA — Number of columns in A. (Input)
Default: NCA = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

NRB — Number of rows in B. (Input)
It must be the same as NRA.
Default: NRB = size (B,1).

NCB — Number of columns in B. (Input)
It must be the same as NCA.
Default: NCB = size (B,2).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

FORTRAN 90 Interface

Generic: CALL CRRCR (A, B [,…])

Specific: The specific interface names are S_CRRCR and D_CRRCR.

FORTRAN 77 Interface

Single: CALL CRRCR (NRA, NCA, A, LDA, NRB, NCB, B, LDB)

Double: The double precision name is DCRRCR.

Example

A 3 × 2 real matrix is copied to a 3 × 2 complex matrix.

```
      USE CRRCR_INT
      USE WRCRN_INT
!                                 Declare variables
      INTEGER    LDA, LDB, NCA, NCB, NRA, NRB
      PARAMETER  (LDA=3, LDB=3, NCA=2, NCB=2, NRA=3, NRB=3)
!
      REAL       A(LDA,NCA)
      COMPLEX    B(LDB,NCB)
!                                 Set values for  A
!                                 A = (  1.0     4.0  )
!                                     (  2.0     5.0  )
!                                     (  3.0     6.0  )
!
      DATA A/1.0, 2.0, 3.0, 4.0, 5.0, 6.0/
!                                 Convert real A to complex B
      CALL CRRCR (A, B)
!                                 Print results
      CALL WRCRN ('B', B)
      END
```

Output

```
                 B
                  1                 2
1  ( 1.000, 0.000)  ( 4.000, 0.000)
2  ( 2.000, 0.000)  ( 5.000, 0.000)
3  ( 3.000, 0.000)  ( 6.000, 0.000)
```

Description

The routine CRRCR copies a real rectangular matrix to a complex rectangular matrix.

CRBCB

Converts a real matrix in band storage mode to a complex matrix in band storage mode.

Required Arguments

A — Real band matrix of order N. (Input)

NLCA — Number of lower codiagonals in A. (Input)

NUCA — Number of upper codiagonals in A. (Input)

***B* —** Complex matrix of order N containing a copy of A. (Output)

NLCB — Number of lower codiagonals in B. (Input)
NLCB must be at least as large as NLCA.

NUCB — Number of upper codiagonals in B. (Input)
NUCB must be at least as large as NUCA.

Optional Arguments

N — Order of the matrices A and B. (Input)
Default: N = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

FORTRAN 90 Interface

Generic: CALL CRBCB (A, NLCA, NUCA, B, NLCB, NUCB [,…])

Specific: The specific interface names are S_CRBCB and D_CRBCB.

FORTRAN 77 Interface

Single: CALL CRBCB (N, A, LDA, NLCA, NUCA, B, LDB, NLCB, NUCB)

Double: The double precision name is DCRBCB.

Example

A real band matrix of order 3 in band storage mode with one upper codiagonal and one lower codiagonal is copied into another complex band matrix in band storage mode.

```
      USE CRBCB_INT
      USE WRCRN_INT
!                                 Declare variables
      INTEGER    LDA, LDB, N, NLCA, NLCB, NUCA, NUCB
      PARAMETER  (LDA=3, LDB=3, N=3, NLCA=1, NLCB=1, NUCA=1, NUCB=1)
!
      REAL       A(LDA,N)
      COMPLEX    B(LDB,N)
!                                 Set values for  A (in band mode)
!                                 A = (  0.0     1.0    1.0)
!                                     (  1.0     1.0    1.0)
!                                     (  1.0     1.0    0.0)
!
      DATA A/0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0/
!                                 Convert real band matrix A
!                                 to complex band matrix B
      CALL CRBCB (A, NLCA, NUCA, B, NLCB, NUCB)
!                                 Print results
      CALL WRCRN ('B', B)
      END
```

Output

```
                            B
                 1                2                3
1  ( 0.000, 0.000)  ( 1.000, 0.000)  ( 1.000, 0.000)
2  ( 1.000, 0.000)  ( 1.000, 0.000)  ( 1.000, 0.000)
3  ( 1.000, 0.000)  ( 1.000, 0.000)  ( 0.000, 0.000)
```

Description

The routine CRBCB converts a real band matrix in band storage mode with NUCA upper codiagonals and NLCA lower codiagonals into a complex band matrix in band storage mode with NUCB upper codiagonals and NLCB lower codiagonals.

CSFRG

Extends a real symmetric matrix defined in its upper triangle to its lower triangle.

Required Arguments

A — N by N symmetric matrix of order N to be filled out. (Input/Output)

Optional Arguments

N – Order of the matrix A. (Input)
Default: N = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

FORTRAN 90 Interface

Generic: CALL CSFRG (A [,…])

Specific: The specific interface names are S_CSFRG and D_CSFRG.

FORTRAN 77 Interface

Single: CALL CSFRG (N, A, LDA)

Double: The double precision name is DCSFRG.

Example

The lower triangular portion of a real 3 × 3 symmetric matrix is filled with the values defined in its upper triangular portion.

```
      USE CSFRG_INT
      USE WRRRN_INT
!                                 Declare variables
      INTEGER    LDA, N
      PARAMETER  (LDA=3, N=3)
!
      REAL       A(LDA,N)
!                                 Set values for  A
!                                 A = (   0.0   3.0   4.0  )
!                                     (         1.0   5.0  )
!                                     (               2.0  )
!
      DATA A/3*0.0, 3.0, 1.0, 0.0, 4.0, 5.0, 2.0/
!                                 Fill the lower portion of A
      CALL CSFRG (A)
!                                 Print results
      CALL WRRRN ('A', A)
      END
```

Output

```
          A
        1       2       3
1   0.000   3.000   4.000
2   3.000   1.000   5.000
3   4.000   5.000   2.000
```

Description

The routine CSFRG converts an $N \times N$ matrix A in symmetric mode into a general matrix by filling in the lower triangular portion of A using the values defined in its upper triangular portion.

CHFCG

Extends a complex Hermitian matrix defined in its upper triangle to its lower triangle.

Required Arguments

A — Complex Hermitian matrix of order N. (Input/Output)
On input, the upper triangle of A defines a Hermitian matrix. On output, the lower triangle of A is defined so that A is Hermitian.

Optional Arguments

N — Order of the matrix. (Input)
Default: N = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

FORTRAN 90 Interface

Generic: CALL CHFCG (A [,…])

Specific: The specific interface names are S_CHFCG and D_CHFCG.

FORTRAN 77 Interface

Single: CALL CHFCG (N, A, LDA)

Double: The double precision name is DCHFCG.

Comments

Informational errors

Type	Code	
3	1	The matrix is not Hermitian. It has a diagonal entry with a small imaginary part.
4	2	The matrix is not Hermitian. It has a diagonal entry with an imaginary part.

Example

A complex 3 × 3 Hermitian matrix defined in its upper triangle is extended to its lower triangle.

```
      USE CHFCG_INT
      USE WRCRN_INT
!                                 Declare variables
      INTEGER    LDA, N
      PARAMETER  (LDA=3, N=3)
!
      COMPLEX    A(LDA,N)
!                                 Set values for  A
!                           A = (  1.0+0.0i  1.0+1.0i  1.0+2.0i  )
!                               (            2.0+0.0i  2.0+2.0i  )
!                               (                      3.0+0.0i  )
!
      DATA A/(1.0,0.0), 2*(0.0,0.0), (1.0,1.0), (2.0,0.0), (0.0,0.0), &
          (1.0,2.0), (2.0,2.0), (3.0,0.0)/
!                                 Fill in lower Hermitian matrix
      CALL CHFCG (A)
!                                 Print results
      CALL WRCRN ('A', A)
      END
```

Output

```
                            A
                 1                2                3
1  ( 1.000, 0.000)  ( 1.000, 1.000)  ( 1.000, 2.000)
2  ( 1.000,-1.000)  ( 2.000, 0.000)  ( 2.000, 2.000)
3  ( 1.000,-2.000)  ( 2.000,-2.000)  ( 3.000, 0.000)
```

Description

The routine CHFCG converts an *N* × *N* complex matrix *A* in Hermitian mode into a complex general matrix by filling in the lower triangular portion of *A* using the values defined in its upper triangular portion.

CSBRB

Copies a real symmetric band matrix stored in band symmetric storage mode to a real band matrix stored in band storage mode.

Required Arguments

A — Real band symmetric matrix of order N. (Input)

NUCA — Number of codiagonals in A. (Input)

B — Real band matrix of order N containing a copy of A. (Output)

NLCB — Number of lower codiagonals in B. (Input)
NLCB must be at least as large as NUCA.

NUCB — Number of upper codiagonals in B. (Input)
NUCB must be at least as large as NUCA.

Optional Arguments

N — Order of the matrices A and B. (Input)
Default: N = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

FORTRAN 90 Interface

Generic: CALL CSBRB (A, NUCA, B, NLCB, NUCB [,…])

Specific: The specific interface names are S_CSBRB and D_CSBRB.

FORTRAN 77 Interface

Single: CALL CSBRB (N, A, LDA, NUCA, B, LDB, NLCB, NUCB)

Double: The double precision name is DCSBRB.

Example

A real matrix of order 4 in band symmetric storage mode with 2 upper codiagonals is copied to a real matrix in band storage mode with 2 upper codiagonals and 2 lower codiagonals.

```
      USE CSBRB_INT
      USE WRRRN_INT
!                                 Declare variables
      INTEGER    LDA, LDB, N, NLCB, NUCA, NUCB
      PARAMETER  (N=4, NUCA=2, LDA=NUCA+1, NLCB=NUCA, NUCB=NUCA, &
                LDB=NLCB+NUCB+1)
!
      REAL       A(LDA,N), B(LDB,N)
!                          Set values for  A, in band mode
!                          A = (  0.0  0.0  2.0  1.0 )
!                              (  0.0  2.0  3.0  1.0 )
!                              (  1.0  2.0  3.0  4.0 )
!
      DATA A/2*0.0, 1.0, 0.0, 2.0, 2.0, 2.0, 3.0, 3.0, 1.0, 1.0, 4.0/
!                                 Copy A to B
      CALL CSBRB (A, NUCA, B, NLCB, NUCB)
!                                 Print results
```

```
      CALL WRRRN ('B', B)
      END
```

Output

```
                  B
          1        2        3        4
1     0.000    0.000    2.000    1.000
2     0.000    2.000    3.000    1.000
3     1.000    2.000    3.000    4.000
4     2.000    3.000    1.000    0.000
5     2.000    1.000    0.000    0.000
```

Description

The routine `CSBRB` copies a real matrix *A* stored in symmetric band mode to a matrix B stored in band mode. The lower codiagonals of *B* are set using the values from the upper codiagonals of *A*.

CHBCB

Copies a complex Hermitian band matrix stored in band Hermitian storage mode to a complex band matrix stored in band storage mode.

Required Arguments

A — Complex band Hermitian matrix of order `N`.   (Input)

NUCA — Number of codiagonals in `A`.   (Input)

B — Complex band matrix of order `N` containing a copy of `A`.   (Output)

NLCB — Number of lower codiagonals in `B`.   (Input)
`NLCB` must be at least as large as `NUCA`.

NUCB — Number of upper codiagonals in `B`.   (Input)
`NUCB` must be at least as large as `NUCA`.

Optional Arguments

N — Order of the matrices `A` and `B`.   (Input)
Default: `N` = size (`A`,2).

LDA — Leading dimension of `A` exactly as specified in the dimension statement of the calling program.   (Input)
Default: `LDA` = size (`A`,1).

LDB — Leading dimension of `B` exactly as specified in the dimension statement of the calling program.   (Input)
Default: `LDB` = size (`B`,1).

FORTRAN 90 Interface

Generic: CALL CHBCB (A, NUCA, B, NLCB, NUCB [,…])

Specific: The specific interface names are S_CHBCB and D_CHBCB.

FORTRAN 77 Interface

Single: CALL CHBCB (N, A, LDA, NUCA, B, LDB, NLCB, NUCB)

Double: The double precision name is DCHBCB.

Comments

Informational errors

Type	Code	
3	1	An element on the diagonal has a complex part that is near zero, the complex part is set to zero.
4	1	An element on the diagonal has a complex part that is not zero.

Example

A complex Hermitian matrix of order 3 in band Hermitian storage mode with one upper codiagonal is copied to a complex matrix in band storage mode.

```
      USE CHBCB_INT
      USE WRCRN_INT
!                                 Declare variables
      INTEGER    LDA, LDB, N, NLCB, NUCA, NUCB
      PARAMETER  (N=3, NUCA=1, LDA=NUCA+1, NLCB=NUCA, NUCB=NUCA, &
                LDB=NLCB+NUCB+1)
!
      COMPLEX    A(LDA,N), B(LDB,N)
!                                 Set values for  A (in band mode)
!                           A = (  0.0+0.0i -1.0+1.0i -2.0+2.0i  )
!                                (  1.0+0.0i  1.0+0.0i  1.0+0.0i  )
!
      DATA A/(0.0,0.0), (1.0,0.0), (-1.0,1.0), (1.0,0.0), (-2.0,2.0), &
          (1.0,0.0)/
!                                 Copy a complex Hermitian band matrix
!                                 to a complex band matrix
      CALL CHBCB (A, NUCA, B, NLCB, NUCB)
!                                 Print results
      CALL WRCRN ('B', B)
      END
```

Output

```
                             B
                 1                  2                  3
1  ( 0.000, 0.000)  (-1.000, 1.000)  (-2.000, 2.000)
2  ( 1.000, 0.000)  ( 1.000, 0.000)  ( 1.000, 0.000)
3  (-1.000,-1.000)  (-2.000,-2.000)  ( 0.000, 0.000)
```

Description

The routine CSBRB copies a complex matrix *A* stored in Hermitian band mode to a matrix *B* stored in complex band mode. The lower codiagonals of *B* are filled using the values in the upper codiagonals of *A*.

TRNRR

Transposes a rectangular matrix.

Required Arguments

A — Real NRA by NCA matrix in full storage mode. (Input)

B — Real NRB by NCB matrix in full storage mode containing the transpose of A. (Output)

Optional Arguments

NRA — Number of rows of A. (Input)
Default: NRA = size (A,1).

NCA — Number of columns of A. (Input)
Default: NCA = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

NRB — Number of rows of B. (Input)
NRB must be equal to NCA.
Default: NRB = size (B,1).

NCB — Number of columns of B. (Input)
NCB must be equal to NRA.
Default: NCB = size (B,2).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

FORTRAN 90 Interface

Generic: CALL TRNRR (A, B [,…])

Specific: The specific interface names are S_TRNRR and D_TRNRR.

FORTRAN 77 Interface

Single: `CALL TRNRR (NRA, NCA, A, LDA, NRB, NCB, B, LDB)`

Double: The double precision name is `DTRNRR`.

Example

Transpose the 5 × 3 real rectangular matrix *A* into the 3 × 5 real rectangular matrix *B*.

```
      USE TRNRR_INT
      USE WRRRN_INT
!                                 Declare variables
      INTEGER    NCA, NCB, NRA, NRB
      PARAMETER  (NCA=3, NCB=5, NRA=5, NRB=3)
!
      REAL       A(NRA,NCA), B(NRB,NCB)
!                                 Set values for A
!                                 A = ( 11.0  12.0  13.0 )
!                                     ( 21.0  22.0  23.0 )
!                                     ( 31.0  32.0  33.0 )
!                                     ( 41.0  42.0  43.0 )
!                                     ( 51.0  52.0  53.0 )
!
      DATA A/11.0, 21.0, 31.0, 41.0, 51.0, 12.0, 22.0, 32.0, 42.0,&
          52.0, 13.0, 23.0, 33.0, 43.0, 53.0/
!                                 B = transpose(A)
      CALL TRNRR (A, B)
!                                 Print results
      CALL WRRRN ('B = trans(A)', B)
      END
```

Output

```
             B = trans(A)
       1        2        3        4        5
1  11.00    21.00    31.00    41.00    51.00
2  12.00    22.00    32.00    42.00    52.00
3  13.00    23.00    33.00    43.00    53.00
```

Comments

If `LDA` = `LDB` and `NRA` = `NCA`, then `A` and `B` can occupy the same storage locations; otherwise, `A` and `B` must be stored separately.

Description

The routine `TRNRR` computes the transpose $B = A^T$ of a real rectangular matrix *A*.

MXTXF

Computes the transpose product of a matrix, $A^T A$.

Required Arguments

A — Real NRA by NCA rectangular matrix. (Input)
The transpose product of A is to be computed.

B — Real NB by NB symmetric matrix containing the transpose product $A^T A$. (Output)

Optional Arguments

NRA — Number of rows in A. (Input)
Default: NRA = size (A,1).

NCA — Number of columns in A. (Input)
Default: NCA = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

NB — Order of the matrix B. (Input)
NB must be equal to NCA.
Default: NB = size (B,1).

LDB — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

FORTRAN 90 Interface

Generic: CALL MXTXF (A, B [,…])

Specific: The specific interface names are S_MXTXF and D_MXTXF.

FORTRAN 77 Interface

Single: CALL MXTXF (NRA, NCA, A, LDA, NB, B, LDB)

Double: The double precision name is DMXTXF.

Example

Multiply the transpose of a 3 × 4 real matrix by itself. The output matrix will be a 4 × 4 real symmetric matrix.

```
      USE MXTXF_INT
      USE WRRRN_INT
!                                 Declare variables
      INTEGER    NB, NCA, NRA
      PARAMETER  (NB=4, NCA=4, NRA=3)
!
      REAL       A(NRA,NCA), B(NB,NB)
!                                 Set values for A
!                                 A = ( 3.0  1.0  4.0  2.0 )
!                                     ( 0.0  2.0  1.0 -1.0 )
!                                     ( 6.0  1.0  3.0  2.0 )
!
      DATA A/3.0, 0.0, 6.0, 1.0, 2.0, 1.0, 4.0, 1.0, 3.0, 2.0, -1.0, &
          2.0/
!                                 Compute B = trans(A)*A
      CALL MXTXF (A, B)
!                                 Print results
      CALL WRRRN (’B = trans(A)*A’, B)
      END
```

Output

```
        B = trans(A)*A
        1       2       3       4
1   45.00    9.00   30.00   18.00
2    9.00    6.00    9.00    2.00
3   30.00    9.00   26.00   13.00
4   18.00    2.00   13.00    9.00
```

Description

The routine `MXTXF` computes the real general matrix $B = A^T A$ given the real rectangular matrix A.

MXTYF

Multiplies the transpose of matrix A by matrix B, $A^T B$.

Required Arguments

A — Real `NRA` by `NCA` matrix. (Input)

B — Real `NRB` by `NCB` matrix. (Input)

C — Real `NCA` by `NCB` matrix containing the transpose product $A^T B$. (Output)

Optional Arguments

NRA — Number of rows in A. (Input)
Default: NRA = size (A,1).

NCA — Number of columns in A. (Input)
Default: NCA = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

NRB — Number of rows in B. (Input)
NRB must be the same as NRA.
Default: NRB = size (B,1).

NCB — Number of columns in B. (Input)
Default: NCB = size (B,2).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

NRC — Number of rows of C. (Input)
NRC must be equal to NCA.
Default: NRC = size (C,1).

NCC — Number of columns of C. (Input)
NCC must be equal to NCB.
Default: NCC = size (C,2).

LDC — Leading dimension of C exactly as specified in the dimension statement of the calling program. (Input)
Default: LDC = size (C,1).

FORTRAN 90 Interface

Generic: `CALL MXTYF (A, B, C [,…])`

Specific: The specific interface names are S_MXTYF and D_MXTYF.

FORTRAN 77 Interface

Single:
```
CALL MXTYF (NRA, NCA, A, LDA, NRB, NCB, B, LDB, NRC, NCC,
            C, LDC)
```

Double: The double precision name is DMXTYF.

Example

Multiply the transpose of a 3 × 4 real matrix by a 3 × 3 real matrix. The output matrix will be a 4 × 3 real matrix.

```
      USE MXTYF_INT
      USE WRRRN_INT
!                                 Declare variables
      INTEGER    NCA, NCB, NCC, NRA, NRB, NRC
      PARAMETER  (NCA=4, NCB=3, NCC=3, NRA=3, NRB=3, NRC=4)
!
      REAL       A(NRA,NCA), B(NRB,NCB), C(NRC,NCC)
!                                 Set values for A
!                                 A = ( 1.0  0.0  2.0  0.0 )
!                                     ( 3.0  4.0 -1.0  0.0 )
!                                     ( 2.0  1.0  2.0  1.0 )
!
!                                 Set values for B
!                                 B = ( -1.0  2.0  0.0 )
!                                     (  3.0  0.0 -1.0 )
!                                     (  0.0  5.0  2.0 )
!
      DATA A/1.0, 3.0, 2.0, 0.0, 4.0, 1.0, 2.0, -1.0, 2.0, 0.0, 0.0, &
          1.0/
      DATA B/-1.0, 3.0, 0.0, 2.0, 0.0, 5.0, 0.0, -1.0, 2.0/
!                                 Compute C = trans(A)*B
      CALL MXTYF (A, B, C)
!                                 Print results
      CALL WRRRN ('C = trans(A)*B', C)
      END
```

Output

```
      C = trans(A)*B
        1        2        3
1    8.00    12.00     1.00
2   12.00     5.00    -2.00
3   -5.00    14.00     5.00
4    0.00     5.00     2.00
```

Description

The routine MXTYF computes the real general matrix $C = A^TB$ given the real rectangular matrices A and B.

MXTYF

Multiplies a matrix A by the transpose of a matrix B, AB^T.

Required Arguments

A — Real NRA by NCA rectangular matrix. (Input)

B — Real NRB by NCB rectangular matrix. (Input)

C — Real NRC by NCC rectangular matrix containing the transpose product AB^T. (Output)

Optional Arguments

NRA — Number of rows in A. (Input)
Default: NRA = size (A,1).

NCA — Number of columns in A. (Input)
Default: NCA = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

NRB — Number of rows in B. (Input)
Default: NRB = size (B,1).

NCB — Number of columns in B. (Input)
NCB must be the same as NCA.
Default: NCB = size (B,2).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

NRC — Number of rows of C. (Input)
NRC must be equal to NRA.
Default: NRC = size (C,1).

NCC — Number of columns of C. (Input)
NCC must be equal to NRB.
Default: NCC = size (C,2).

LDC — Leading dimension of C exactly as specified in the dimension statement of the calling program. (Input)
Default: LDC = size (C,1).

FORTRAN 90 Interface

Generic: CALL MXYTF (A, B, C [,…])

Specific: The specific interface names are S_MXYTF and D_MXYTF.

FORTRAN 77 Interface

Single: `CALL MXYTF (NRA, NCA, A, LDA, NRB, NCB, B, LDB, NRC, NCC, C, LDC)`

Double: The double precision name is `DMXYTF`.

Example

Multiply a 3 × 4 real matrix by the transpose of a 3 × 4 real matrix. The output matrix will be a 3 × 3 real matrix.

```
      USE MXYTF_INT
      USE WRRRN_INT
!                                 Declare variables
      INTEGER    NCA, NCB, NCC, NRA, NRB, NRC
      PARAMETER  (NCA=4, NCB=4, NCC=3, NRA=3, NRB=3, NRC=3)
!
      REAL       A(NRA,NCA), B(NRB,NCB), C(NRC,NCC)
!                                 Set values for A
!                                 A = ( 1.0  0.0  2.0  0.0 )
!                                     ( 3.0  4.0 -1.0  0.0 )
!                                     ( 2.0  1.0  2.0  1.0 )
!
!                                 Set values for B
!                                 B = ( -1.0  2.0  0.0  2.0 )
!                                     (  3.0  0.0 -1.0 -1.0 )
!                                     (  0.0  5.0  2.0  5.0 )
!
      DATA A/1.0, 3.0, 2.0, 0.0, 4.0, 1.0, 2.0, -1.0, 2.0, 0.0, 0.0, &
          1.0/
      DATA B/-1.0, 3.0, 0.0, 2.0, 0.0, 5.0, 0.0, -1.0, 2.0, 2.0, -1.0, &
          5.0/
!                                 Compute C = A*trans(B)
      CALL MXYTF (A, B, C)
!                                 Print results
      CALL WRRRN ('C = A*trans(B)', C)
      END
```

Output

```
      C = A*trans(B)
        1       2       3
1   -1.00    1.00    4.00
2    5.00   10.00   18.00
3    2.00    3.00   14.00
```

Description

The routine `MXYTF` computes the real general matrix $C = AB^T$ given the real rectangular matrices A and B.

MRRRR

Multiplies two real rectangular matrices, *AB*.

Required Arguments

A — Real NRA by NCA matrix in full storage mode. (Input)

B — Real NRB by NCB matrix in full storage mode. (Input)

C — Real NRC by NCC matrix containing the product *AB* in full storage mode. (Output)

Optional Arguments

NRA — Number of rows of A. (Input)
Default: NRA = size (A,1).

NCA — Number of columns of A. (Input)
Default: NCA = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

NRB — Number of rows of B. (Input)
NRB must be equal to NCA.
Default: NRB = size (B,1).

NCB — Number of columns of B. (Input)
Default: NCB = size (B,2).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

NRC — Number of rows of C. (Input)
NRC must be equal to NRA.
Default: NRC = size (C,1).

NCC — Number of columns of C. (Input)
NCC must be equal to NCB.
Default: NCC = size (C,2).

LDC — Leading dimension of C exactly as specified in the dimension statement of the calling program. (Input)
Default: LDC = size (C,1).

FORTRAN 90 Interface

Generic: `CALL MRRRR (A, B, C [,…])`

Specific: The specific interface names are `S_MRRRR` and `D_MRRRR`.

FORTRAN 77 Interface

Single: `CALL MRRRR (NRA, NCA, A, LDA, NRB, NCB, B, LDB, NRC, NCC, C, LDC)`

Double: The double precision name is `DMRRRR`.

Example

Multiply a 3 × 4 real matrix by a 4 × 3 real matrix. The output matrix will be a 3 × 3 real matrix.

```
      USE MRRRR_INT
      USE WRRRN_INT
!                                 Declare variables
      INTEGER    NCA, NCB, NCC, NRA, NRB, NRC
      PARAMETER  (NCA=4, NCB=3, NCC=3, NRA=3, NRB=4, NRC=3)
!
      REAL       A(NRA,NCA), B(NRB,NCB), C(NRC,NCC)
!                                 Set values for A
!                                 A = ( 1.0  0.0  2.0  0.0 )
!                                     ( 3.0  4.0 -1.0  0.0 )
!                                     ( 2.0  1.0  2.0  1.0 )
!
!                                 Set values for B
!                                 B = ( -1.0  0.0  2.0 )
!                                     (  3.0  5.0  2.0 )
!                                     (  0.0  0.0 -1.0 )
!                                     (  2.0 -1.0  5.0 )
!
      DATA A/1.0, 3.0, 2.0, 0.0, 4.0, 1.0, 2.0, -1.0, 2.0, 0.0, 0.0, &
          1.0/
      DATA B/-1.0, 3.0, 0.0, 2.0, 0.0, 5.0, 0.0, -1.0, 2.0, 2.0, -1.0, &
          5.0/
!                                 Compute C = A*B
      CALL MRRRR (A, B, C)
!                                 Print results
      CALL WRRRN (’C = A*B’, C)
      END
```

Output

```
          C = A*B
        1       2       3
1   -1.00    0.00    0.00
2    9.00   20.00   15.00
3    3.00    4.00    9.00
```

Description

Given the real rectangular matrices A and B, MRRRR computes the real rectangular matrix $C = AB$.

MCRCR

Multiplies two complex rectangular matrices, AB.

Required Arguments

A — Complex NRA by NCA rectangular matrix. (Input)

B — Complex NRB by NCB rectangular matrix. (Input)

C — Complex NRC by NCC rectangular matrix containing the product A * B. (Output)

Optional Arguments

NRA — Number of rows of A. (Input)
Default: NRA = size (A,1).

NCA — Number of columns of A. (Input)
Default: NCA = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

NRB — Number of rows of B. (Input)
NRB must be equal to NCA.
Default: NRB = size (B,1).

NCB — Number of columns of B. (Input)
Default: NCB = size (B,2).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

NRC — Number of rows of C. (Input)
NRC must be equal to NRA.
Default: NRC = size (C,1).

NCC — Number of columns of C. (Input)
NCC must be equal to NCB.
Default: NCC = size (C,2).

LDC — Leading dimension of C exactly as specified in the dimension statement of the calling program. (Input)
Default: LDC = size (C,1).

FORTRAN 90 Interface

Generic: CALL MCRCR (A, B, C [,…])

Specific: The specific interface names are S_MCRCR and D_MCRCR.

FORTRAN 77 Interface

Single: CALL MCRCR (NRA, NCA, A, LDA, NRB, NCB, B, LDB, NRC, NCC, C, LDC)

Double: The double precision name is DMCRCR.

Example

Multiply a 3 × 4 complex matrix by a 4 × 3 complex matrix. The output matrix will be a 3 × 3 complex matrix.

```
      USE MCRCR_INT
      USE WRCRN_INT
!                                 Declare variables
      INTEGER    NCA, NCB, NCC, NRA, NRB, NRC
      PARAMETER  (NCA=4, NCB=3, NCC=3, NRA=3, NRB=4, NRC=3)
!
      COMPLEX    A(NRA,NCA), B(NRB,NCB), C(NRC,NCC)
!                                 Set values for A
!             A = ( 1.0 + 1.0i  -1.0+ 2.0i  0.0 + 1.0i  0.0 - 2.0i )
!                 ( 3.0 + 7.0i  6.0 - 4.0i  2.0 - 1.0i  0.0 + 1.0i )
!                 ( 1.0 + 0.0i  1.0 - 2.0i  -2.0+ 0.0i  0.0 + 0.0i )
!
!                                 Set values for B
!             B = ( 2.0 + 1.0i  3.0 + 2.0i  3.0 + 1.0i )
!                 ( 2.0 - 1.0i  4.0 - 2.0i  5.0 - 3.0i )
!                 ( 1.0 + 0.0i  0.0 - 1.0i  0.0 + 1.0i )
!                 ( 2.0 + 1.0i  1.0 + 2.0i  0.0 - 1.0i )
!
      DATA A/(1.0,1.0), (3.0,7.0), (1.0,0.0), (-1.0,2.0), (6.0,-4.0), &
          (1.0,-2.0), (0.0,1.0), (2.0,-1.0), (-2.0,0.0), (0.0,-2.0), &
          (0.0,1.0), (0.0,0.0)/
      DATA B/(2.0,1.0), (2.0,-1.0), (1.0,0.0), (2.0,1.0), (3.0,2.0), &
          (4.0,-2.0), (0.0,-1.0), (1.0,2.0), (3.0,1.0), (5.0,-3.0), &
          (0.0,1.0), (0.0,-1.0)/
!                                 Compute C = A*B
      CALL MCRCR (A, B, C)
!                                 Print results
      CALL WRCRN ('C = A*B', C)
      END
```

Output

```
                     C = A*B
                 1                  2                  3
1  (  3.00,  5.00)  (  6.00, 13.00)  (  0.00, 17.00)
2  (  8.00,  4.00)  (  8.00, -2.00)  ( 22.00,-12.00)
3  (  0.00, -4.00)  (  3.00, -6.00)  (  2.00,-14.00)
```

Description

Given the complex rectangular matrices *A* and *B*, MCRCR computes the complex rectangular matrix *C* = *AB*.

HRRRR

Computes the Hadamard product of two real rectangular matrices.

Required Arguments

A — Real NRA by NCA rectangular matrix. (Input)

B — Real NRB by NCB rectangular matrix. (Input)

C — Real NRC by NCC rectangular matrix containing the Hadamard product of A and B. (Output)
If A is not needed, then C can share the same storage locations as A. Similarly, if B is not needed, then C can share the same storage locations as B.

Optional Arguments

NRA — Number of rows of A. (Input)
Default: NRA = size (A,1).

NCA — Number of columns of A. (Input)
Default: NCA = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

NRB — Number of rows of B. (Input)
NRB must be equal to NRA.
Default: NRB = size (B,1).

NCB — Number of columns of B. (Input)
NCB must be equal to NCA.
Default: NCB = size (B,2).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

NRC — Number of rows of C. (Input)
NRC must be equal to NRA.
Default: NRC = size (C,1).

NCC — Number of columns of C. (Input)
NCC must be equal to NCA.
Default: NCC = size (C,2).

LDC — Leading dimension of C exactly as specified in the dimension statement of the calling program. (Input)
Default: LDC = size (C,1).

FORTRAN 90 Interface

Generic: CALL HRRRR (A, B, C [,…])

Specific: The specific interface names are S_HRRRR and D_HRRRR.

FORTRAN 77 Interface

Single: CALL HRRRR (NRA, NCA, A, LDA, NRB, NCB, B, LDB, NRC, NCC, C, LDC)

Double: The double precision name is DHRRRR.

Example

Compute the Hadamard product of two 4 × 4 real matrices. The output matrix will be a 4 × 4 real matrix.

```
      USE HRRRR_INT
      USE WRRRN_INT
!                                 Declare variables
      INTEGER    NCA, NCB, NCC, NRA, NRB, NRC
      PARAMETER  (NCA=4, NCB=4, NCC=4, NRA=4, NRB=4, NRC=4)
!
      REAL       A(NRA,NCA), B(NRB,NCB), C(NRC,NCC)
!                                 Set values for A
!                                 A = ( -1.0  0.0 -3.0  8.0 )
!                                     (  2.0  1.0  7.0  2.0 )
!                                     (  3.0 -2.0  2.0 -6.0 )
!                                     (  4.0  1.0 -5.0 -8.0 )
!
!                                 Set values for B
!                                 B = (  2.0  3.0  0.0 -10.0 )
!                                     (  1.0 -1.0  4.0   2.0 )
!                                     ( -1.0 -2.0  7.0   1.0 )
!                                     (  2.0  1.0  9.0   0.0 )
!
      DATA A/-1.0, 2.0, 3.0, 4.0, 0.0, 1.0, -2.0, 1.0, -3.0, 7.0, 2.0, &
          -5.0, 8.0, 2.0, -6.0, -8.0/
      DATA B/2.0, 1.0, -1.0, 2.0, 3.0, -1.0, -2.0, 1.0, 0.0, 4.0, 7.0, &
          9.0, -10.0, 2.0, 1.0, 0.0/
!                                 Compute Hadamard product of A and B
      CALL HRRRR (A, B, C)
!                                 Print results
      CALL WRRRN ('C = A (*) B', C)
      END
```

Output

```
            C = A (*) B
        1       2       3       4
1   -2.00    0.00    0.00  -80.00
2    2.00   -1.00   28.00    4.00
3   -3.00    4.00   14.00   -6.00
4    8.00    1.00  -45.00    0.00
```

Description

The routine HRRRR computes the Hadamard product of two real matrices *A* and *B* and returns a real matrix *C*, where $C_{ij} = A_{ij}B_{ij}$.

BLINF

This function computes the bilinear form $x^T Ay$.

Function Return Value

BLINF — The value of $x^T Ay$ is returned in BLINF.   (Output)

Required Arguments

A — Real NRA by NCA matrix.   (Input)

X — Real vector of length NRA.   (Input)

Y — Real vector of length NCA.   (Input)

Optional Arguments

NRA — Number of rows of A.   (Input)
Default: NRA = size (A, 1).

NCA — Number of columns of A.   (Input)
Default: NCA = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

FORTRAN 90 Interface

Generic: BLINF (A, X, Y [,…])

Specific: The specific interface names are S_BLINF and D_BLINF.

FORTRAN 77 Interface

Single: BLINF(NRA, NCA, A, LDA, X, Y)

Double: The double precision name is DBLINF.

Example

Compute the bilinear form $x^T Ay$, where x is a vector of length 5, A is a 5 × 2 matrix and y is a vector of length 2.

```
      USE BLINF_INT
      USE UMACH_INT
!                                 Declare variables
      INTEGER    NCA, NRA
      PARAMETER  (NCA=2, NRA=5)
!
      INTEGER    NOUT
      REAL       A(NRA,NCA), VALUE, X(NRA), Y(NCA)
!                                 Set values for A
!                                 A = ( -2.0  2.0 )
!                                     (  3.0 -6.0 )
!                                     ( -4.0  7.0 )
!                                     (  1.0 -8.0 )
!                                     (  0.0 10.0 )
!                                 Set values for X
!                                 X = (  1.0 -2.0  3.0 -4.0 -5.0 )
!                                 Set values for Y
!                                 Y = ( -6.0  3.0 )
!
      DATA A/-2.0, 3.0, -4.0, 1.0, 0.0, 2.0, -6.0, 7.0, -8.0, 10.0/
      DATA X/1.0, -2.0, 3.0, -4.0, -5.0/
      DATA Y/-6.0, 3.0/
!                                 Compute bilinear form
      VALUE = BLINF(A,X,Y)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,*) ’ The bilinear form trans(x)*A*y = ’, VALUE
      END
```

Output

```
The bilinear form trans(x)*A*y =     195.000
```

Comments

The quadratic form can be computed by calling `BLINF` with the vector `X` in place of the vector `Y`.

Description

Given the real rectangular matrix *A* and two vectors *x* and *y*, `BLINF` computes the bilinear form $x^T Ay$.

POLRG

Evaluates a real general matrix polynomial.

Required Arguments

A — `N` by `N` matrix for which the polynomial is to be computed. (Input)

COEF — Vector of length `NCOEF` containing the coefficients of the polynomial in order of increasing power. (Input)

B — `N` by `N` matrix containing the value of the polynomial evaluated at `A`. (Output)

Optional Arguments

N — Order of the matrix `A`. (Input)
Default: `N` = size (`A`,1).

LDA — Leading dimension of `A` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDA` = size (`A`,1).

NCOEF — Number of coefficients. (Input)
Default: `NCOEF` = size (`COEF`,1).

LDB — Leading dimension of `B` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDB` = size (`B`,1).

FORTRAN 90 Interface

Generic: `CALL POLRG (A, COEF, B [,…])`

Specific: The specific interface names are `S_POLRG` and `D_POLRG`.

FORTRAN 77 Interface

Single: `CALL POLRG (N, A, LDA, NCOEF, COEF, B, LDB)`

Double: The double precision name is DPOLRG.

Example

This example evaluates the matrix polynomial $3I + A + 2A^2$, where A is a 3 × 3 matrix.

```
      USE POLRG_INT
      USE WRRRN_INT
!                                 Declare variables
      INTEGER    LDA, LDB, N, NCOEF
      PARAMETER  (N=3, NCOEF=3, LDA=N, LDB=N)
!
      REAL       A(LDA,N), B(LDB,N), COEF(NCOEF)
!                                 Set values of A and COEF
!
!                                 A = (  1.0    3.0    2.0  )
!                                     ( -5.0    1.0    7.0  )
!                                     (  1.0    5.0   -4.0  )
!
!                                 COEF = (3.0, 1.0, 2.0)
!
      DATA A/1.0, -5.0, 1.0, 3.0, 1.0, 5.0, 2.0, 7.0, -4.0/
      DATA COEF/3.0, 1.0, 2.0/
!
!                                 Evaluate B = 3I + A + 2*A**2
      CALL POLRG (A, COEF, B)
!                                 Print B
      CALL WRRRN ('B = 3I + A + 2*A**2', B)
      END
```

Output

```
    B = 3I + A + 2*A**2
         1       2       3
1   -20.0    35.0    32.0
2   -11.0    46.0   -55.0
3   -55.0   -19.0   105.0
```

Comments

Workspace may be explicitly provided, if desired, by use of P2LRG/DP2LRG. The reference is

```
CALL P2LRG (N, A, LDA, NCOEF, COEF, B, LDB, WORK)
```

The additional argument is

WORK — Work vector of length N * N.

Description

Let m = NCOEF and c = COEF.

The routine POLRG computes the matrix polynomial

$$B = \sum_{k=1}^{m} c_k A^{k-1}$$

using Horner's scheme

$$B = \left(\ldots \left(\left(c_m A + c_{m-1} I \right) A + c_{m-2} I \right) A + \ldots + c_1 I \right)$$

where I is the $N \times N$ identity matrix.

MURRV

Multiplies a real rectangular matrix by a vector.

Required Arguments

A — Real NRA by NCA rectangular matrix. (Input)

X — Real vector of length NX. (Input)

Y — Real vector of length NY containing the product A * X if IPATH is equal to 1 and the product trans(A) * X if IPATH is equal to 2. (Output)

Optional Arguments

NRA — Number of rows of A. (Input)
Default: NRA = size (A,1).

NCA — Number of columns of A. (Input)
Default: NCA = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

NX — Length of the vector X. (Input)
NX must be equal to NCA if IPATH is equal to 1. NX must be equal to NRA if IPATH is equal to 2.
Default: NX = size (X,1).

IPATH — Integer flag. (Input)
IPATH = 1 means the product Y = A * X is computed. IPATH = 2 means the product Y = trans(A) * X is computed, where trans(A) is the transpose of A.
Default: IPATH =1.

NY — Length of the vector Y. (Input)
NY must be equal to NRA if IPATH is equal to 1. NY must be equal to NCA if IPATH is

equal to 2.
Default: NY = size (Y,1).

FORTRAN 90 Interface

Generic: CALL MURRV (A, X, Y [,…])

Specific: The specific interface names are S_MURRV and D_MURRV.

FORTRAN 77 Interface

Single: CALL MURRV (NRA, NCA, A, LDA, NX, X, IPATH, NY, Y)

Double: The double precision name is DMURRV.

Example

Multiply a 3 × 3 real matrix by a real vector of length 3. The output vector will be a real vector of length 3.

```
      USE MURRV_INT
      USE WRRRN_INT
!                                 Declare variables
      INTEGER    LDA, NCA, NRA, NX, NY
      PARAMETER  (NCA=3, NRA=3, NX=3, NY=3)
!
      INTEGER    IPATH
      REAL       A(NRA,NCA), X(NX), Y(NY)
!                                 Set values for A and X
!                                 A = ( 1.0  0.0  2.0 )
!                                     ( 0.0  3.0  0.0 )
!                                     ( 4.0  1.0  2.0 )
!
!                                 X = ( 1.0  2.0  1.0 )
!
!
      DATA A/1.0, 0.0, 4.0, 0.0, 3.0, 1.0, 2.0, 0.0, 2.0/
      DATA X/1.0, 2.0, 1.0/
!                                 Compute y = Ax
      IPATH = 1
      CALL MURRV (A, X, Y)
!                                 Print results
      CALL WRRRN ('y = Ax', Y, 1, NY, 1)
      END
```

Output

```
        y = Ax
    1       2       3
3.000   6.000   8.000
```

Description

If IPATH = 1, MURRV computes $y = Ax$, where A is a real general matrix and x and y are real vectors. If IPATH = 2, MURRV computes $y = A^T x$.

MURBV

Multiplies a real band matrix in band storage mode by a real vector.

Required Arguments

A — Real NLCA + NUCA + 1 by N band matrix stored in band mode. (Input)

NLCA — Number of lower codiagonals in A. (Input)

NUCA — Number of upper codiagonals in A. (Input)

X — Real vector of length NX. (Input)

Y — Real vector of length NY containing the product A * X if IPATH is equal to 1 and the product trans(A) * X if IPATH is equal to 2. (Output)

Optional Arguments

N — Order of the matrix. (Input)
Default: N = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

NX — Length of the vector X. (Input)
NX must be equal to N.
Default: NX = size (X,1).

IPATH — Integer flag. (Input)
IPATH = 1 means the product Y = A * X is computed. IPATH = 2 means the product Y = trans(A) * X is computed, where trans(A) is the transpose of A.
Default: IPATH = 1.

NY — Length of vector Y. (Input)
NY must be equal to N.
Default: NY = size (Y,1).

FORTRAN 90 Interface

Generic: `CALL MURBV (A, NLCA, NUCA, X, Y [,…])`

Specific: The specific interface names are S_MURBV and D_MURBV.

FORTRAN 77 Interface

Single: CALL MURBV (N, A, LDA, NLCA, NUCA, NX, X, IPATH, NY, Y)

Double: The double precision name is DMURBV.

Example

Multiply a real band matrix of order 6, with two upper codiagonals and two lower codiagonals stored in band mode, by a real vector of length 6. The output vector will be a real vector of length 6.

```
      USE MURBV_INT
      USE WRRRN_INT
!                                 Declare variables
      INTEGER    LDA, N, NLCA, NUCA, NX, NY
      PARAMETER  (LDA=5, N=6, NLCA=2, NUCA=2, NX=6, NY=6)
!
      INTEGER    IPATH
      REAL       A(LDA,N), X(NX), Y(NY)
!                                 Set values for A (in band mode)
!                                 A = ( 0.0  0.0  1.0  2.0  3.0  4.0 )
!                                     ( 0.0  1.0  2.0  3.0  4.0  5.0 )
!                                     ( 1.0  2.0  3.0  4.0  5.0  6.0 )
!                                     (-1.0 -2.0 -3.0 -4.0 -5.0  0.0 )
!                                     (-5.0 -6.0 -7.0 -8.0  0.0  0.0 )
!
!                                 Set values for X
!                                 X = (-1.0  2.0 -3.0  4.0 -5.0  6.0 )
!
      DATA A/0.0, 0.0, 1.0, -1.0, -5.0, 0.0, 1.0, 2.0, -2.0, -6.0, &
          1.0, 2.0, 3.0, -3.0, -7.0, 2.0, 3.0, 4.0, -4.0, -8.0, 3.0, &
          4.0, 5.0, -5.0, 0.0, 4.0, 5.0, 6.0, 0.0, 0.0/
      DATA X/-1.0, 2.0, -3.0, 4.0, -5.0, 6.0/
!                                 Compute y = Ax
      IPATH = 1
      CALL MURBV (A, NLCA, NUCA, X, Y)
!                                 Print results
      CALL WRRRN ('y = Ax', Y, 1, NY, 1)
      END
```

Output

```
                      y = Ax
    1        2        3        4        5        6
-2.00     7.00   -11.00    17.00    10.00    29.00
```

Description

If IPATH = 1, MURBV computes $y = Ax$, where A is a real band matrix and x and y are real vectors. If IPATH = 2, MURBV computes $y = A^T x$.

MUCRV

Multiplies a complex rectangular matrix by a complex vector.

Required Arguments

A — Complex NRA by NCA rectangular matrix.   (Input)

X — Complex vector of length NX.   (Input)

Y — Complex vector of length NY containing the product A * X if IPATH is equal to 1 and the product trans(A) * X if IPATH is equal to 2.   (Output)

Optional Arguments

NRA — Number of rows of A.   (Input)
Default: NRA = size (A,1).

NCA — Number of columns of A.   (Input)
Default: NCA = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program.   (Input)
Default: LDA = size (A,1).

NX — Length of the vector X.   (Input)
NX must be equal to NCA if IPATH is equal to 1. NX must be equal to NRA if IPATH is equal to 2.
Default: NX = size (X,1).

IPATH — Integer flag.   (Input)
IPATH = 1 means the product Y = A * X is computed. IPATH = 2 means the product Y = trans(A) * X is computed, where trans(A) is the transpose of A.
Default: IPATH =1.

NY — Length of the vector Y.   (Input)
NY must be equal to NRA if IPATH is equal to 1. NY must be equal to NCA if IPATH is equal to 2.
Default: NY = size (Y,1).

FORTRAN 90 Interface

Generic: CALL MUCRV (A, X, Y [,…])

Specific: The specific interface names are S_MUCRV and D_MUCRV.

FORTRAN 77 Interface

Single: CALL MUCRV (NRA, NCA, A, LDA, NX, X, IPATH, NY, Y)

Double: The double precision name is DMUCRV.

Example

Multiply a 3 × 3 complex matrix by a complex vector of length 3. The output vector will be a complex vector of length 3.

```
      USE MUCRV_INT
      USE WRCRN_INT
!                                 Declare variables
      INTEGER    NCA, NRA, NX, NY
      PARAMETER  (NCA=3, NRA=3, NX=3, NY=3)
!
      INTEGER    IPATH
      COMPLEX    A(NRA,NCA), X(NX), Y(NY)
!
!                                 Set values for A and X
!             A = ( 1.0 + 2.0i  3.0 + 4.0i  1.0 + 0.0i )
!                 ( 2.0 + 1.0i  3.0 + 2.0i  0.0 - 1.0i )
!                 ( 2.0 - 1.0i  1.0 + 0.0i  0.0 + 1.0i )
!
!             X = ( 1.0 - 1.0i  2.0 - 2.0i  0.0 - 1.0i )
!
      DATA A/(1.0,2.0), (2.0,1.0), (2.0,-1.0), (3.0,4.0), (3.0,2.0), &
          (1.0,0.0), (1.0,0.0), (0.0,-1.0), (0.0,1.0)/
      DATA X/(1.0,-1.0), (2.0,-2.0), (0.0,-1.0)/
!                                 Compute y = Ax
      IPATH = 1
      CALL MUCRV (A, X, Y)
!                                 Print results
      CALL WRCRN ('y = Ax', Y, 1, NY, 1)
      END
```

Output

```
                     y = Ax
              1                2                3
( 17.00,  2.00)  ( 12.00, -3.00)  (  4.00, -5.00)
```

Description

If IPATH = 1, MUCRV computes $y = Ax$, where A is a complex general matrix and x and y are complex vectors. If IPATH = 2, MUCRV computes $y = A^T x$.

MUCBV

Multiplies a complex band matrix in band storage mode by a complex vector.

Required Arguments

A — Complex NLCA + NUCA + 1 by N band matrix stored in band mode. (Input)

NLCA — Number of lower codiagonals in A. (Input)

NUCA — Number of upper codiagonals in A. (Input)

X — Complex vector of length NX. (Input)

Y — Complex vector of length NY containing the product A * X if IPATH is equal to 1 and the product trans(A) * X if IPATH is equal to 2. (Output)

Optional Arguments

N — Order of the matrix. (Input)
Default: N = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

NX — Length of the vector X. (Input)
NX must be equal to N.
Default: NX = size (X,1).

IPATH — Integer flag. (Input)
IPATH = 1 means the product Y = A * X is computed. IPATH = 2 means the product Y = trans(A) * X is computed, where trans(A) is the transpose of A.
Default: IPATH = 1.

NY — Length of vector Y. (Input)
NY must be equal to N.
Default: NY = size (Y,1).

FORTRAN 90 Interface

Generic: CALL MUCBV (A, NLCA, NUCA, X, Y [,…])

Specific: The specific interface names are S_MUCBV and D_MUCBV.

FORTRAN 77 Interface

Single: CALL MUCBV (N, A, LDA, NLCA, NUCA, NX, X, IPATH, NY, Y)

Double: The double precision name is DMUCBV.

Example

Multiply the transpose of a complex band matrix of order 4, with one upper codiagonal and two lower codiagonals stored in band mode, by a complex vector of length 3. The output vector will be a complex vector of length 3.

```
      USE MUCBV_INT
      USE WRCRN_INT
!                                 Declare variables
      INTEGER    LDA, N, NLCA, NUCA, NX, NY
      PARAMETER  (LDA=4, N=4, NLCA=2, NUCA=1, NX=4, NY=4)
!
      INTEGER    IPATH
      COMPLEX    A(LDA,N), X(NX), Y(NY)
!                                   Set values for A (in band mode)
!            A = (  0.0+ 0.0i   1.0+ 2.0i   3.0+ 4.0i   5.0+ 6.0i )
!                ( -1.0- 1.0i  -1.0- 1.0i  -1.0- 1.0i  -1.0- 1.0i )
!                ( -1.0+ 2.0i  -1.0+ 3.0i  -2.0+ 1.0i   0.0+ 0.0i )
!                (  2.0+ 0.0i   0.0+ 2.0i   0.0+ 0.0i   0.0+ 0.0i )
!
!                                  Set values for X
!            X = ( 3.0 + 4.0i  0.0 + 0.0i  1.0 + 2.0i  -2.0 - 1.0i )
!
      DATA A/(0.0,0.0), (-1.0,-1.0), (-1.0,2.0), (2.0,0.0), (1.0,2.0), &
          (-1.0,-1.0), (-1.0,3.0), (0.0,2.0), (3.0,4.0), (-1.0,-1.0), &
          (-2.0,1.0), (0.0,0.0), (5.0,6.0), (-1.0,-1.0), (0.0,0.0), &
          (0.0,0.0)/
      DATA X/(3.0,4.0), (0.0,0.0), (1.0,2.0), (-2.0,-1.0)/
!                                 Compute y = Ax
      IPATH = 2
      CALL MUCBV (A, NLCA, NUCA, X, Y, IPATH=IPATH)
!                                 Print results
      CALL WRCRN ('y = Ax', Y, 1, NY, 1)
      END
```

Output

```
                             y = Ax
              1                2                3                4
(  3.00, -3.00)  (-10.00,  7.00)  (  6.00, -3.00)  ( -6.00, 19.00)
```

Description

If IPATH = 1, MUCBV computes $y = Ax$, where A is a complex band matrix and x and y are complex vectors. If IPATH = 2, MUCBV computes $y = A^T x$.

ARBRB

Adds two band matrices, both in band storage mode.

Required Arguments

A — N by N band matrix with NLCA lower codiagonals and NUCA upper codiagonals stored in band mode with dimension (NLCA + NUCA + 1) by N. (Input)

NLCA — Number of lower codiagonals of A. (Input)

NUCA — Number of upper codiagonals of A. (Input)

B — N by N band matrix with NLCB lower codiagonals and NUCB upper codiagonals stored in band mode with dimension (NLCB + NUCB + 1) by N. (Input)

NLCB — Number of lower codiagonals of B. (Input)

NUCB — Number of upper codiagonals of B. (Input)

C — N by N band matrix with NLCC lower codiagonals and NUCC upper codiagonals containing the sum A + B in band mode with dimension (NLCC + NUCC + 1) by N. (Output)

NLCC — Number of lower codiagonals of C. (Input)
NLCC must be at least as large as max(NLCA, NLCB).

NUCC — Number of upper codiagonals of C. (Input)
NUCC must be at least as large as max(NUCA, NUCB).

Optional Arguments

N — Order of the matrices A, B and C. (Input)
Default: N = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

LDC — Leading dimension of C exactly as specified in the dimension statement of the calling program. (Input)
Default: LDC = size (C,1).

FORTRAN 90 Interface

Generic:
```
CALL ARBRB (A, NLCA, NUCA, B, NLCB, NUCB, C, NLCC,
           NUCC [,…])
```

Specific: The specific interface names are S_ARBRB and D_ARBRB.

FORTRAN 77 Interface

Single: CALL ARBRB (N, A, LDA, NLCA, NUCA, B, LDB, NLCB, NUCB, C, LDC, NLCC, NUCC)

Double: The double precision name is DARBRB.

Example

Add two real matrices of order 4 stored in band mode. Matrix *A* has one upper codiagonal and one lower codiagonal. Matrix *B* has no upper codiagonals and two lower codiagonals. The output matrix *C*, has one upper codiagonal and two lower codiagonals.

```
      USE ARBRB_INT
      USE WRRRN_INT
!                                 Declare variables
      INTEGER    LDA, LDB, LDC, N, NLCA, NLCB, NLCC, NUCA, NUCB, NUCC
      PARAMETER  (LDA=3, LDB=3, LDC=4, N=4, NLCA=1, NLCB=2, NLCC=2, &
                NUCA=1, NUCB=0, NUCC=1)
!
      REAL       A(LDA,N), B(LDB,N), C(LDC,N)
!                                 Set values for  A (in band mode)
!                                 A = (  0.0     2.0    3.0   -1.0)
!                                     (  1.0     1.0    1.0    1.0)
!                                     (  0.0     3.0    4.0    0.0)
!
!                                 Set values for  B (in band mode)
!                                 B = (  3.0     3.0    3.0    3.0)
!                                     (  1.0    -2.0    1.0    0.0)
!                                     ( -1.0     2.0    0.0    0.0)
!
      DATA A/0.0, 1.0, 0.0, 2.0, 1.0, 3.0, 3.0, 1.0, 4.0, -1.0, 1.0, &
          0.0/
      DATA B/3.0, 1.0, -1.0, 3.0, -2.0, 2.0, 3.0, 1.0, 0.0, 3.0, 0.0, &
          0.0/
!                                 Add A and B to obtain C (in band
!                                                          mode)
      CALL ARBRB (A, NLCA, NUCA, B, NLCB, NUCB, C, NLCC, NUCC)
!                                 Print results
      CALL WRRRN (’C = A+B’, C)
      END
```

Output

```
             C = A+B
        1       2       3       4
1   0.000   2.000   3.000  -1.000
2   4.000   4.000   4.000   4.000
3   1.000   1.000   5.000   0.000
4  -1.000   2.000   0.000   0.000
```

Description

The routine ARBRB adds two real matrices stored in band mode, returning a real matrix stored in band mode.

ACBCB

Adds two complex band matrices, both in band storage mode.

Required Arguments

A — N by N complex band matrix with NLCA lower codiagonals and NUCA upper codiagonals stored in band mode with dimension (NLCA + NUCA + 1) by N. (Input)

NLCA — Number of lower codiagonals of A. (Input)

NUCA — Number of upper codiagonals of A. (Input)

B — N by N complex band matrix with NLCB lower codiagonals and NUCB upper codiagonals stored in band mode with dimension (NLCB + NUCB + 1) by N. (Input)

NLCB — Number of lower codiagonals of B. (Input)

NUCB — Number of upper codiagonals of B. (Input)

C — N by N complex band matrix with NLCC lower codiagonals and NUCC upper codiagonals containing the sum A + B in band mode with dimension (NLCC + NUCC + 1) by N. (Output)

NLCC — Number of lower codiagonals of C. (Input)
NLCC must be at least as large as max(NLCA, NLCB).

NUCC — Number of upper codiagonals of C. (Input)
NUCC must be at least as large as max(NUCA, NUCB).

Optional Arguments

N — Order of the matrices A, B and C. (Input)
Default: N = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

LDB — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)
Default: LDB = size (B,1).

LDC — Leading dimension of C exactly as specified in the dimension statement of the calling program. (Input)
Default: LDC = size (C,1).

FORTRAN 90 Interface

Generic: `CALL ACBCB (A, NLCA, NUCA, B, NLCB, NUCB, C, NLCC, NUCC [,…])`

Specific: The specific interface names are `S_ACBCB` and `D_ACBCB`.

FORTRAN 77 Interface

Single: `CALL ACBCB (N, A, LDA, NLCA, NUCA, B, LDB, NLCB, NUCB, C, LDC, NLCC, NUCC)`

Double: The double precision name is `DACBCB`.

Example

Add two complex matrices of order 4 stored in band mode. Matrix *A* has two upper codiagonals and no lower codiagonals. Matrix *B* has no upper codiagonals and two lower codiagonals. The output matrix *C* has two upper codiagonals and two lower codiagonals.

```
      USE ACBCB_INT
      USE WRCRN_INT
!                                 Declare variables
      INTEGER    LDA, LDB, LDC, N, NLCA, NLCB, NLCC, NUCA, NUCB, NUCC
      PARAMETER  (LDA=3, LDB=3, LDC=5, N=3, NLCA=0, NLCB=2, NLCC=2, &
                 NUCA=2, NUCB=0, NUCC=2)
!
      COMPLEX    A(LDA,N), B(LDB,N), C(LDC,N)
!                                 Set values for A (in band mode)
!                A = ( 0.0 + 0.0i  0.0 + 0.0i  3.0 - 2.0i )
!                    ( 0.0 + 0.0i  -1.0+ 3.0i  6.0 + 0.0i )
!                    ( 1.0 + 4.0i  5.0 - 2.0i  3.0 + 1.0i )
!
!                                 Set values for B (in band mode)
!                B = ( 3.0 + 1.0i  4.0 + 1.0i  7.0 - 1.0i )
!                    ( -1.0- 4.0i  9.0 + 3.0i  0.0 + 0.0i )
!                    ( 2.0 - 1.0i  0.0 + 0.0i  0.0 + 0.0i )
!
      DATA A/(0.0,0.0), (0.0,0.0), (1.0,4.0), (0.0,0.0), (-1.0,3.0), &
          (5.0,-2.0), (3.0,-2.0), (6.0,0.0), (3.0,1.0)/
      DATA B/(3.0,1.0), (-1.0,-4.0), (2.0,-1.0), (4.0,1.0), (9.0,3.0), &
          (0.0,0.0), (7.0,-1.0), (0.0,0.0), (0.0,0.0)/
!                                 Compute C = A+B
      CALL ACBCB (A, NLCA, NUCA, B, NLCB, NUCB, C, NLCC, NUCC)
!                                 Print results
      CALL WRCRN ('C = A+B', C)
      END
```

Output

```
                    C = A+B
              1                 2                 3
1  (  0.00,  0.00)  (  0.00,  0.00)  (  3.00, -2.00)
2  (  0.00,  0.00)  ( -1.00,  3.00)  (  6.00,  0.00)
```

```
3  (  4.00,  5.00)  (  9.00, -1.00)  ( 10.00,  0.00)
4  ( -1.00, -4.00)  (  9.00,  3.00)  (  0.00,  0.00)
5  (  2.00, -1.00)  (  0.00,  0.00)  (  0.00,  0.00)
```

Description

The routine ACBCB adds two complex matrices stored in band mode, returning a complex matrix stored in band mode.

NRIRR

Computes the infinity norm of a real matrix.

Required Arguments

A — Real NRA by NCA matrix whose infinity norm is to be computed. (Input)

ANORM — Real scalar containing the infinity norm of A. (Output)

Optional Arguments

NRA — Number of rows of A. (Input)
Default: NRA = size (A,1).

NCA — Number of columns of A. (Input)
Default: NCA = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

FORTRAN 90 Interface

Generic: CALL NRIRR (A, ANORM [,…])

Specific: The specific interface names are S_NRIRR and D_NRIRR.

FORTRAN 77 Interface

Single: CALL NRIRR (NRA, NCA, A, LDA, ANORM)

Double: The double precision name is DNRIRR.

Example

Compute the infinity norm of a 3 × 4 real rectangular matrix.

```
      USE NRIRR_INT
      USE UMACH_INT
!                                 Declare variables
      INTEGER    NCA, NRA
      PARAMETER  (NCA=4, NRA=3)
!
      INTEGER    NOUT
      REAL       A(NRA,NCA), ANORM
!
!                                 Set values for A
!                                 A = ( 1.0  0.0  2.0  0.0 )
!                                     ( 3.0  4.0 -1.0  0.0 )
!                                     ( 2.0  1.0  2.0  1.0 )
!
      DATA A/1.0, 3.0, 2.0, 0.0, 4.0, 1.0, 2.0, -1.0, 2.0, 0.0, 0.0, &
          1.0/
!                                 Compute the infinity norm of A
      CALL NRIRR (A, ANORM)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,*) ' The infinity norm of A is ', ANORM
      END
```

Output

```
The infinity norm of A is     8.00000
```

Description

The routine NRIRR computes the infinity norm of a real rectangular matrix *A*. If *m* = NRA and *n* = NCA, then the ∞-norm of *A* is

$$\left\|A\right\|_\infty = \max_{1\le i\le m}\sum_{j=1}^{n}\left|A_{ij}\right|$$

This is the maximum of the sums of the absolute values of the row elements.

NR1RR

Computes the 1-norm of a real matrix.

Required Arguments

A — Real NRA by NCA matrix whose 1-norm is to be computed. (Input)

ANORM — Real scalar containing the 1-norm of A. (Output)

Optional Arguments

NRA — Number of rows of A. (Input)
Default: NRA = size (A,1).

NCA — Number of columns of A. (Input)
Default: NCA = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

FORTRAN 90 Interface

Generic: CALL NR1RR (A, ANORM [,…])

Specific: The specific interface names are S_NR1RR and D_NR1RR.

FORTRAN 77 Interface

Single: CALL NR1RR (NRA, NCA, A, LDA, ANORM)

Double: The double precision name is DNR1RR.

Example

Compute the 1-norm of a 3 × 4 real rectangular matrix.

```
      USE NR1RR_INT
      USE UMACH_INT
!                                Declare variables
      INTEGER    NCA, NRA
      PARAMETER  (NCA=4, NRA=3)
!
      INTEGER    NOUT
      REAL       A(NRA,NCA), ANORM
!
!                Set values for A
!                        A = ( 1.0  0.0  2.0  0.0 )
!                            ( 3.0  4.0 -1.0  0.0 )
!                            ( 2.0  1.0  2.0  1.0 )
!
      DATA A/1.0, 3.0, 2.0, 0.0, 4.0, 1.0, 2.0, -1.0, 2.0, 0.0, 0.0, &
          1.0/
!                                    Compute the L1 norm of A
      CALL NR1RR (A, ANORM)
!                                    Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,*) ’ The 1-norm of A is ’, ANORM
      END
```

Output

```
The 1-norm of A is     6.00000
```

Description

The routine NR1RR computes the 1-norm of a real rectangular matrix A. If m = NRA and n = NCA, then the 1-norm of A is

$$\|A\|_1 = \max_{1 \le j \le n} \sum_{i=1}^{m} \left| A_{ij} \right|$$

This is the maximum of the sums of the absolute values of the column elements.

NR2RR

Computes the Frobenius norm of a real rectangular matrix.

Required Arguments

A — Real NRA by NCA rectangular matrix. (Input)

ANORM — Frobenius norm of A. (Output)

Optional Arguments

NRA — Number of rows of A. (Input)
Default: NRA = size (A,1).

NCA — Number of columns of A. (Input)
Default: NCA = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

FORTRAN 90 Interface

Generic: CALL NR2RR (A, ANORM [,…])

Specific: The specific interface names are S_NR2RR and D_NR2RR.

FORTRAN 77 Interface

Single: CALL NR2RR (NRA, NCA, A, LDA, ANORM)

Double: The double precision name is DNR2RR.

Example

Compute the Frobenius norm of a 3 × 4 real rectangular matrix.

```
      USE NR2RR_INT
      USE UMACH_INT
!                                 Declare variables
      INTEGER    LDA, NCA, NRA
      PARAMETER  (LDA=3, NCA=4, NRA=3)
!
      INTEGER    NOUT
      REAL       A(LDA,NCA), ANORM
!
!                                 Set values for A
!                                 A = ( 1.0  0.0  2.0  0.0 )
!                                     ( 3.0  4.0 -1.0  0.0 )
!                                     ( 2.0  1.0  2.0  1.0 )
!
      DATA A/1.0, 3.0, 2.0, 0.0, 4.0, 1.0, 2.0, -1.0, 2.0, 0.0, 0.0, &
          1.0/
!
!                                 Compute Frobenius norm of A
      CALL NR2RR (A, ANORM)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,*) ' The Frobenius norm of A is ', ANORM
      END
```

Output

```
The Frobenius norm of A is     6.40312
```

Description

The routine NR2RR computes the Frobenius norm of a real rectangular matrix A. If m = NRA and n = NCA, then the Frobenius norm of A is

$$\|A\|_2 = \left[\sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij}^2 \right]^{1/2}$$

NR1RB

Computes the 1-norm of a real band matrix in band storage mode.

Required Arguments

A — Real (NUCA + NLCA + 1) by N array containing the N by N band matrix in band storage mode. (Input)

NLCA — Number of lower codiagonals of A. (Input)

NUCA — Number of upper codiagonals of A. (Input)

ANORM — Real scalar containing the 1-norm of A. (Output)

Optional Arguments

N — Order of the matrix. (Input)
Default: N = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

FORTRAN 90 Interface

Generic: CALL NR1RB (A, NLCA, NUCA, ANORM [,…])

Specific: The specific interface names are S_NR1RB and D_NR1RB.

FORTRAN 77 Interface

Single: CALL NR1RB (N, A, LDA, NLCA, NUCA, ANORM)

Double: The double precision name is DNR1RB.

Example

Compute the 1-norm of a 4 × 4 real band matrix stored in band mode.

```
      USE NR1RB_INT
      USE UMACH_INT
!                                 Declare variables
      INTEGER    LDA, N, NLCA, NUCA
      PARAMETER  (LDA=4, N=4, NLCA=2, NUCA=1)
!
      INTEGER    NOUT
      REAL       A(LDA,N), ANORM
!
!                                 Set values for A (in band mode)
!                                 A = (  0.0  2.0  2.0  3.0  )
!                                     ( -2.0 -3.0 -4.0 -1.0  )
!                                     (  2.0  1.0  0.0  0.0  )
!                                     (  0.0  1.0  0.0  0.0  )
!
      DATA A/0.0, -2.0, 2.0, 0.0, 2.0, -3.0, 1.0, 1.0, 2.0, -4.0, 0.0, &
          0.0, 3.0, -1.0, 2*0.0/
!                                 Compute the L1 norm of A
      CALL NR1RB (A, NLCA, NUCA, ANORM)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,*) ’ The 1-norm of A is ’, ANORM
      END
```

Output

```
The 1-norm of A is     7.00000
```

Description

The routine NR1RB computes the 1-norm of a real band matrix *A*. The 1-norm of a matrix *A* is

$$\|A\|_1 = \max_{1 \le j \le N} \sum_{i=1}^{N} \left|A_{ij}\right|$$

This is the maximum of the sums of the absolute values of the column elements.

NR1CB

Computes the 1-norm of a complex band matrix in band storage mode.

Required Arguments

A — Complex (NUCA + NLCA + 1) by N array containing the N by N band matrix in band storage mode. (Input)

NLCA — Number of lower codiagonals of A. (Input)

NUCA — Number of upper codiagonals of A. (Input)

ANORM — Real scalar containing the 1-norm of A. (Output)

Optional Arguments

N — Order of the matrix. (Input)
Default: N = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A,1).

FORTRAN 90 Interface

Generic: CALL NR1CB (A, NLCA, NUCA, ANORM [,…])

Specific: The specific interface names are S_NR1CB and D_NR1CB.

FORTRAN 77 Interface

Single: CALL NR1CB (N, A, LDA, NLCA, NUCA, ANORM)

Double: The double precision name is DNR1CB.

Example

Compute the 1-norm of a complex matrix of order 4 in band storage mode.

```
      USE NR1CB_INT
      USE UMACH_INT
!                                  Declare variables
      INTEGER    LDA, N, NLCA, NUCA
      PARAMETER  (LDA=4, N=4, NLCA=2, NUCA=1)
!
      INTEGER    NOUT
      REAL       ANORM
      COMPLEX    A(LDA,N)
!
!                                  Set values for A (in band mode)
!                      A = (  0.0+0.0i  2.0+3.0i -1.0+1.0i -2.0-1.0i )
!                          ( -2.0+3.0i  1.0+0.0i -4.0-1.0i  0.0-4.0i )
!                          (  2.0+2.0i  4.0+6.0i  3.0+2.0i  0.0+0.0i )
!                          (  0.0-1.0i  2.0+1.0i  0.0+0.0i  0.0+0.0i )
!
      DATA A/(0.0,0.0), (-2.0,3.0), (2.0,2.0), (0.0,-1.0), (2.0,3.0), &
          (1.0,0.0), (4.0,6.0), (2.0,1.0), (-1.0,1.0), (-4.0,-1.0), &
          (3.0,2.0), (0.0,0.0), (-2.0,-1.0), (0.0,-4.0), (0.0,0.0), &
          (0.0,0.0)/
!                                  Compute the L1 norm of A
      CALL NR1CB (A, NLCA, NUCA, ANORM)
!                                  Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,*) ’ The 1-norm of A is ’, ANORM
      END
```

Output

```
The 1-norm of A is     19.0000
```

Description

The routine `NR1CB` computes the 1-norm of a complex band matrix *A*. The 1-norm of a complex matrix *A* is

$$\|A\|_1 = \max_{1\le j\le N}\sum_{i=1}^{N}\left[\left|\Re A_{ij}\right|+\left|\Im A_{ij}\right|\right]$$

DISL2

This function computes the Euclidean (2-norm) distance between two points.

Function Return Value

DISL2 — Euclidean (2-norm) distance between the points `X` and `Y`.   (Output)

Required Arguments

X — Vector of length max(N * |INCX|, 1). (Input)

Y — Vector of length max(N * |INCY|, 1). (Input)

Optional Arguments

N — Length of the vectors X and Y. (Input)
Default: N = size (X,1).

INCX — Displacement between elements of X. (Input)
The I-th element of X is X(1 + (I – 1) * INCX) if INCX is greater than or equal to zero or X(1 + (I – N) * INCX) if INCX is less than zero.
Default: INCX = 1.

INCY — Displacement between elements of Y. (Input)
The I-th element of Y is Y(1 + (I – 1) * INCY) if INCY is greater than or equal to zero or Y(1 + (I – N) * INCY) if INCY is less than zero.
Default: INCY = 1.

FORTRAN 90 Interface

Generic: DISL2 (X, Y [,…])

Specific: The specific interface names are S_DISL2 and D_DISL2.

FORTRAN 77 Interface

Single: DISL2(N, X, INCX, Y, INCY)

Double: The double precision function name is DDISL2.

Example

Compute the Euclidean (2-norm) distance between two vectors of length 4.

```
      USE DISL2_INT
      USE UMACH_INT
!                                 Declare variables
      INTEGER    INCX, INCY, N
      PARAMETER  (N=4)
!
      INTEGER    NOUT
      REAL       VAL, X(N), Y(N)
!
!                                 Set values for X and Y
!                                 X = ( 1.0 -1.0  0.0  2.0 )
!
!                                 Y = ( 4.0  2.0  1.0 -3.0 )
```

```
!
      DATA X/1.0, -1.0, 0.0, 2.0/
      DATA Y/4.0, 2.0, 1.0, -3.0/
!                                 Compute L2 distance
      VAL = DISL2(X,Y)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,*) ’ The 2-norm distance is ’, VAL
      END
```

Output

```
The 2-norm distance is     6.63325
```

Description

The function `DISL2` computes the Euclidean (2-norm) distance between two points *x* and *y*. The Euclidean distance is defined to be

$$\left[\sum_{i=1}^{N}\left(x_i - y_i\right)^2\right]^{1/2}$$

DISL1

This function computes the 1-norm distance between two points.

Function Return Value

DISL1 — 1-norm distance between the points `X` and `Y`. (Output)

Required Arguments

X — Vector of length max(`N` * |`INCX`|, 1). (Input)

Y — Vector of length max(`N` * |`INCY`|, 1). (Input)

Optional Arguments

N — Length of the vectors `X` and `Y`. (Input)
Default: `N` = size (`X`,1).

INCX — Displacement between elements of `X`. (Input)
The `I`-th element of `X` is `X`(1 + (`I` – 1) * `INCX`) if `INCX` is greater than or equal to zero or `X`(1 + (`I` – `N`) * `INCX`) if `INCX` is less than zero.
Default: `INCX` = 1.

INCY — Displacement between elements of `Y`. (Input)
The `I`-th element of `Y` is `Y`(1 + (`I` – 1) * `INCY`) if `INCY` is greater than or equal to zero

or Y(1 + (I − N) * INCY) if INCY is less than zero.
Default: INCY = 1.

FORTRAN 90 Interface

Generic: DISL1 (X, Y [,…])

Specific: The specific interface names are S_DISL1 and D_DISL1.

FORTRAN 77 Interface

Single: DISL1(N, X, INCX, Y, INCY)

Double: The double precision function name is DDISL1.

Example

Compute the 1-norm distance between two vectors of length 4.

```
      USE DISL1_INT
      USE UMACH_INT
!                                 Declare variables
      INTEGER    INCX, INCY, N
      PARAMETER  (N=4)
!
      INTEGER    NOUT
      REAL       VAL, X(N), Y(N)
!
!                                 Set values for X and Y
!                                 X = ( 1.0 -1.0  0.0  2.0 )
!
!                                 Y = ( 4.0  2.0  1.0 -3.0 )
!
      DATA X/1.0, -1.0, 0.0, 2.0/
      DATA Y/4.0, 2.0, 1.0, -3.0/
!                                 Compute L1 distance
      VAL = DISL1(X,Y)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,*) ' The 1-norm distance is ', VAL
      END
```

Output

```
The 1-norm distance is     12.0000
```

Description

The function DISL1 computes the 1-norm distance between two points x and y. The 1-norm distance is defined to be

$$\sum_{i=1}^{N} \left| x_i - y_i \right|$$

DISLI

This function computes the infinity norm distance between two points.

Function Return Value

DISLI — Infinity norm distance between the points X and Y. (Output)

Required Arguments

X — Vector of length max(N * |INCX|, 1). (Input)

Y — Vector of length max(N * |INCY|, 1). (Input)

Optional Arguments

N — Length of the vectors X and Y. (Input)
Default: N = size (X,1).

INCX — Displacement between elements of X. (Input)
The I-th element of X is X(1 + (I – 1) *INCX) if INCX is greater than or equal to zero or X(1 + (I – N) * INCX) if INCX is less than zero.
Default: INCX = 1.

INCY — Displacement between elements of Y. (Input)
The I-th element of Y is Y(1 + (I – 1) * INCY) if INCY is greater than or equal to zero or Y(1 + (I – N) * INCY) if INCY is less than zero.
Default: INCY = 1.

FORTRAN 90 Interface

Generic: DISLI (X, Y [,…])

Specific: The specific interface names are S_DISLI and D_DISLI.

FORTRAN 77 Interface

Single: DISLI(N, X, INCX, Y, INCY)

Double: The double precision function function name is DDISLI.

Example

Compute the ∞-norm distance between two vectors of length 4.

```
      USE DISLI_INT
      USE UMACH_INT
!                                 Declare variables
      INTEGER    INCX, INCY, N
      PARAMETER  (N=4)
!
      INTEGER    NOUT
      REAL       VAL, X(N), Y(N)
!
!                                 Set values for X and Y
!                                 X = ( 1.0 -1.0  0.0  2.0 )
!
!                                 Y = ( 4.0  2.0  1.0 -3.0 )
!
      DATA X/1.0, -1.0, 0.0, 2.0/
      DATA Y/4.0, 2.0, 1.0, -3.0/
!                                 Compute L-infinity distance
      VAL = DISLI(X,Y)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,*) ’ The infinity-norm distance is ’, VAL
      END
```

Output

```
The infinity-norm distance is     5.00000
```

Description

The function DISLI computes the 1-norm distance between two points x and y. The 1norm distance is defined to be

$$\max_{1 \le i \le N} \left| x_i - y_i \right|$$

VCONR

Computes the convolution of two real vectors.

Required Arguments

X — Vector of length NX. (Input)

Y — Vector of length NY. (Input)

Z — Vector of length NZ containing the convolution Z = X * Y. (Output)

Optional Arguments

NX — Length of the vector X. (Input)
Default: NX = size (X,1).

NY — Length of the vector Y. (Input)
Default: NY = size (Y,1).

NZ — Length of the vector Z. (Input)
NZ must be at least NX + NY – 1.
Default: NZ = size (Z,1).

FORTRAN 90 Interface

Generic: CALL VCONR (X, Y, Z [,…])

Specific: The specific interface names are S_VCONR and D_VCONR.

FORTRAN 77 Interface

Single: CALL VCONR (NX, X, NY, Y, NZ, Z)

Double: The double precision name is DVCONR.

Example

In this example, the convolution of a vector *x* of length 8 and a vector *y* of length 3 is computed. The resulting vector *z* is of length 8 + 3 – 1 = 10. (The vector *y* is sometimes called a *filter*.)

```
      USE VCONR_INT
      USE WRRRN_INT
      INTEGER    NX, NY, NZ
      PARAMETER  (NX=8, NY=3, NZ=NX+NY-1)
!
      REAL       X(NX), Y(NY), Z(NZ)
!                                 Set values for X
!                       X = (1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0)
!                                 Set values for Y
!                       Y = (0.0  0.0  1.0)
!
      DATA X/1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0/
      DATA Y/0.0, 0.0, 1.0/
!                                 Compute vector convolution
!                                 Z = X * Y
      CALL VCONR (X,Y,Z)
!                                 Print results
      CALL WRRRN ('Z = X (*) Y', Z, 1, NZ, 1)
      END
```

Output

```
                                   Z = X (*) Y
    1        2        3          4         5         6         7         8         9        10
0.000   0.000   1.000    2.000    3.000    4.000    5.000    6.000    7.000    8.000
```

Comments

Workspace may be explicitly provided, if desired, by use of V2ONR/DV2ONR. The reference is

```
CALL V2ONR (NX, X, NY, Y, NZ, Z, XWK, YWK, ZWK, WK)
```

The additional arguments are as follows:

XWK — Complex work array of length NX + NY – 1.

YWK — Complex work array of length NX + NY – 1.

ZWK — Complex work array of length NX + NY – 1.

WK — Real work array of length 6 * (NX + NY – 1) + 15.

Description

The routine VCONR computes the convolution z of two real vectors x and y. Let n_x = NX, n_y = NY and n_z = NZ. The vector z is defined to be

$$z_j = \sum_{k=1}^{n_x} x_{j-k+1} y_k \qquad \text{for } j = 1, 2, \ldots, n_z$$

where $n_z = n_x + n_y - 1$. If the index $j - k + 1$ is outside the range $1, 2, \ldots, n_x$, then x_{j-k+1} is taken to be zero.

The fast Fourier transform is used to compute the convolution. Define the complex vector u of length $n_z = n_x + n_y - 1$ to be

$$u = \left(x_1, x_2, \ldots, x_{n_x}, 0, \ldots, 0\right)$$

The complex vector v, also of length n_z, is defined similarly using y. Then, by the Fourier convolution theorem,

$$\hat{w}_i = \hat{u}_i \hat{v}_i \qquad \text{for } i = 1, 2, \ldots, n_z$$

where the $\hat{u}$ indicates the Fourier transform of u computed via IMSL routine FFTCF (see Chapter 6, Transforms) IMSL routine FFTCB (see Chapter 6, Transforms) is used to compute the complex vector w from $\hat{w}$. The vector z is then found by taking the real part of the vector w.

VCONC

Computes the convolution of two complex vectors.

Required Arguments

X — Complex vector of length NX. (Input)

Y — Complex vector of length NY. (Input)

Z — Complex vector of length NZ containing the convolution Z = X * Y. (Output)

Optional Arguments

NX — Length of the vector X. (Input)
Default: NX = size (X,1).

NY — Length of the vector Y. (Input)
Default: NY = size (Y,1).

NZ — Length of the vector Z. (Input)
NZ must be at least NX + NY – 1.
Default: NZ = size (Z,1).

FORTRAN 90 Interface

Generic: CALL VCONC (X, Y, Z [,…])

Specific: The specific interface names are S_VCONC and D_VCONC.

FORTRAN 77 Interface

Single: CALL VCONC (NX, X, NY, Y, NZ, Z)

Double: The double precision name is DVCONC.

Example

In this example, the convolution of a vector *x* of length 4 and a vector *y* of length 3 is computed. The resulting vector *z* is of length 4 + 3 –*y* is sometimes called a *filter*.)

```
      USE VCONC_INT
      USE WRCRN_INT
      INTEGER    NX, NY, NZ
      PARAMETER  (NX=4, NY=3, NZ=NX+NY-1)
!
      COMPLEX    X(NX), Y(NY), Z(NZ)
!                                 Set values for X
!                X = ( 1.0+2.0i 3.0+4.0i 5.0+6.0i 7.0+8.0i )
!                                 Set values for Y
!                Y = (0.0+0i 0.0+0i 1.0+0i )
!
      DATA X/(1.0,2.0), (3.0,4.0), (5.0,6.0), (7.0,8.0)/
      DATA Y/(0.0,0.0), (0.0,0.0), (1.0,1.0)/
```

```
!                                  Compute vector convolution
!                                  Z = X * Y
      CALL VCONC (X,Y,Z)
!                                  Print results
      CALL WRCRN ('Z = X (*) Y', Z, 1, NZ, 1)
      END
```

Output

```
                          Z = X (*) Y
             1                  2                  3                  4
(  0.00,  0.00)   (  0.00,  0.00)   ( -1.00,  3.00)   ( -1.00,  7.00)

             5                  6
( -1.00, 11.00)   ( -1.00, 15.00)
```

Comments

Workspace may be explicitly provided, if desired, by use of V2ONC/DV2ONC. The reference is

`CALL V2ONC (NX, X, NY, Y, NZ, Z, XWK, YWK, WK)`

The additional arguments are as follows:

XWK — Complex work array of length NX + NY – 1.

YWK — Complex work array of length NX + NY – 1.

WK — Real work arrary of length 6 * (NX + NY –1) + 15.

Description

The routine VCONC computes the convolution z of two complex vectors x and y. Let n_x = NX, then n_y = NY and n_z = NZ. The vector z is defined to be

$$z_j = \sum_{k=1}^{n_x} x_{j-k+1} y_k \qquad \text{for } j = 1, 2, \ldots, n_z$$

where $n_z = n_x + n_y - 1$. If the index $j - k + 1$ is outside the range 1, 2, …, n_x, then x_{j-k+1} is taken to be zero.

The fast Fourier transform is used to compute the convolution. Define the complex vector u of length $n_z = n_x + n_y - 1$ to be

$$u = \left(x_1, x_2, \ldots, x_{n_z}, 0, \ldots, 0\right)$$

The complex vector v, also of length n_z, is defined similarly using y. Then, by the Fourier convolution theorem,

$$\hat{z}_i = \hat{u}_i \hat{v}_i \qquad \text{for } i = 1, 2, \ldots, n_z$$

where the $\hat{u}$ indicates the Fourier transform of u computed using IMSL routine FFTCF (see Chapter 6, Transforms). The complex vector z is computed from $\hat{w}$ via IMSL routine FFTCB (see Chapter 6, Transforms).

Extended Precision Arithmetic

This section describes a set of routines for mixed precision arithmetic. The routines are designed to allow the computation and use of the full quadruple precision result from the multiplication of two double precision numbers. An array called the accumulator stores the result of this multiplication. The result of the multiplication is added to the current contents of the accumulator. It is also possible to add a double precision number to the accumulator or to store a double precision approximation in the accumulator.

The mixed double precision arithmetic routines are described below. The accumulator array, QACC, is a double precision array of length 2. Double precision variables are denoted by DA and DB. Available operations are:

Initialize a real accumulator, QACC ← DA.

```
CALL DQINI (DA, QACC)
```

Store a real accumulator, DA ← QACC.

```
CALL DQSTO (QACC, DA)
```

Add to a real accumulator, QACC ← QACC + DA.

```
CALL DQADD (DA, QACC)
```

Add a product to a real accumulator, QACC ← QACC + DA*DB.

```
CALL DQMUL (DA, DB, QACC)
```

There are also mixed double complex arithmetic versions of the above routines. The accumulator, ZACC, is a double precision array of length 4. Double complex variables are denoted by ZA and ZB. Available operations are:

Initialize a complex accumulator, ZACC ← ZA.

```
CALL ZQINI (ZA, ZACC)
```

Store a complex accumulator, ZA ← ZACC.

```
CALL ZQSTO (ZACC, ZA)
```

Add to a complex accumulator, ZACC ← ZACC + ZA.

```
CALL ZQADD (ZA, ZACC)
```

Add a product to a complex accumulator, ZACC ← ZACC + ZA * ZB.

```
CALL ZQMUL (ZA, ZB, ZACC)
```

Example

In this example, the value of 1.0D0/3.0D0 is computed in quadruple precision using Newton's method. Four iterations of

$$x_{k+1} = x_k + \left(x_k - ax_k^2\right)$$

with $a = 3$ are taken. The error $ax - 1$ is then computed. The results are accurate to approximately twice the usual double precision accuracy, as given by the IMSL routine DMACH(4), in the

Reference Material section of this manual. Since DMACH is machine dependent, the actual accuracy obtained is also machine dependent.

```
      USE IMSL_LIBRARIES
      INTEGER    I, NOUT
      DOUBLE PRECISION A, DACC(2), DMACH, ERROR, SACC(2), X(2), X1, X2, EPSQ
!
      CALL UMACH (2, NOUT)
      A = 3.0D0
      CALL DQINI (1.0001D0/A, X)
!                                 Compute X(K+1) = X(K) - A*X(K)*X(K)
!                                 + X(K)
      DO 10  I=1, 4
         X1 = X(1)
         X2 = X(2)
!                                 Compute X + X
         CALL DQADD (X1, X)
         CALL DQADD (X2, X)
!                                 Compute X*X
         CALL DQINI (0.0D0, DACC)
         CALL DQMUL (X1, X1, DACC)
         CALL DQMUL (X1, X2, DACC)
         CALL DQMUL (X1, X2, DACC)
         CALL DQMUL (X2, X2, DACC)
!                                 Compute -A*(X*X)
         CALL DQINI (0.0D0, SACC)
         CALL DQMUL (-A, DACC(1), SACC)
         CALL DQMUL (-A, DACC(2), SACC)
!                                 Compute -A*(X*X) + (X + X)
         CALL DQADD (SACC(1), X)
         CALL DQADD (SACC(2), X)
   10 CONTINUE
!                                 Compute A*X - 1
      CALL DQINI (0.0D0, SACC)
      CALL DQMUL (A, X(1), SACC)
      CALL DQMUL (A, X(2), SACC)
      CALL DQADD (-1.0D0, SACC)
      CALL DQSTO (SACC, ERROR)
!                                 ERROR should be less than MACHEPS**2
      EPSQ = AMACH(4)
      EPSQ = EPSQ * EPSQ
      WRITE (NOUT,99999) ERROR, ERROR/EPSQ
!
99999 FORMAT (’  A*X - 1 = ’, D15.7, ’ = ’, F10.5, ’*MACHEPS**2’)
      END
```

Output

```
A*X - 1 =   0.6162976D-32 =    0.12500*MACHEPS**2
```

Chapter 10: Linear Algebra Operators and Generic Functions

Routines

MPI REQUIRED

Introduction

MPI REQUIRED

This chapter describes numerical linear algebra software packaged as operations that are executed with a function notation similar to standard mathematics. The resulting interface is a great simplification. It alters the way libraries are presented to the user. Many computations of numerical linear algebra are documented here as operators and generic functions. A notation is developed reminiscent of matrix algebra. This allows the Fortran 90 user to express mathematical formulas in terms of operators. Thus, important aspects of "object-oriented" programming are provided as a part of this chapter's design.

A comprehensive Fortran 90 module, *linear_operators*, defines the operators and functions. Its use provides this simplification. Subroutine calls and the use of type-dependent procedure names are largely avoided. This makes a rapid development cycle possible, at least for the purposes of experiments and proof-of-concept. The goal is to provide the Fortran 90 programmer with an interface, operators, and functions that are useful and succinct. The modules can be used with existing Fortran programs, but the operators provide a more readable program. Frequently this approach requires more hidden working storage. The size of the executable program may be larger than alternatives using subroutines. There are applications wherein the operator and function interface does not have the functionality that is available using subroutine libraries. To retain greater flexibility, some users will continue to require the traditional techniques of calling subroutines.

A parallel computation for many of the defined operators and functions has been implemented. Most of the detailed communication is hidden from the user. Those functions having this data type computed in parallel are marked in **`bold type`**. The section "Parallelism Using MPI" (in this chapter) gives an introduction on how users should write their codes to use other machines on a network.

Matrix Algebra Operations

Consider a Fortran 90 code fragment that solves a linear system of algebraic equations, $Ay = b$, then computes the residual $r = b - Ay$. A standard mathematical notation is often used to write the solution,

$$y = A^{-1}b$$

A user thinks: "matrix and right-hand side yields solution." The code shows the computation of this mathematical solution using a defined Fortran operator "**`.ix.`**", and random data obtained with the function, *rand*. This operator is read "inverse matrix times." The residuals are computed with another defined Fortran operator "**`.x.`**", read "matrix times vector." Once a user understands the equivalence of a mathematical formula with the corresponding Fortran operator, it is possible to write this program with little effort. The last line of the example before `end` is discussed below.

```
USE linear_operators
   integer,parameter :: n=3; real A(n,n), y(n), b(n), r(n)
   A=rand(A); b=rand(b); y = A .ix. b
   r = b - (A .x. y )
end
```

The IMSL Fortran Library provides additional lower-level software that implements the operation "**.ix**.", the function *rand*, matrix multiply "**.x.**", and others not used in this example. Standard matrix products and inverse operations of matrix algebra are shown in the following table:

Defined Array Operation	Matrix Operation	Alternative in Fortran 90
A .x. B	AB	matmul(A, B)
.i. A	A^{-1}	lin_sol_gen lin_sol_lsq

.t. A, .h. A	A^T, A^H	transpose(A)
		conjg(transpose(A))

A .ix. B	$A^{-1}B$	lin_sol_gen lin_sol_lsq
B .xi. A	BA^{-1}	lin_sol_gen lin_sol_lsq

A .tx. B, or (.t. A) .x. B
A .hx. B, or (.h. A) .x. B
	$A^T B, A^H B$	matmul(transpose (A), B)
		matmul(conjg(transpose(A)), B)

B .xt. A, or B .x. (.t. A)
B .xh. A, or B .x. (.h. A)
	BA^T, BA^H	matmul(B, transpose(A))
		matmul(B, conjg(transpose(A)))

Operators apply generically to all precisions and floating-point data types and to objects that are broader in scope than arrays. For example, the matrix product "**.x.**." applies to matrix times vector and matrix times matrix represented as Fortran 90 arrays. It also applies to "independent matrix products." For this, use the notion: *a box of problems* to refer to independent linear algebra computations, of the same kind and dimension, but different data. The *racks* of the box are the distinct problems. In terms of Fortran 90 arrays, a rank-3, assumed-shape array is the data structure used for a box. The first two dimensions are the data for a matrix problem; the third dimension is the rack number. Each problem is independent of other problems in consecutive racks of the box. We use parallelism of an underlying network of processors when computing these disjoint problems.

In addition to the operators **.ix.**, **.xi.**, **.i.**, and **.x.**, additional operators .t., .h., **.tx.**, **.hx.**,

`.xt.,` and **`.xh.`** are provided for complex matrices. Since the transpose matrix is defined for complex matrices, this meaning is kept for the defined operations. In order to write one defined operation for both real and complex matrices, use the conjugate-transpose in all cases. This will result in only real operations when the data arrays are real.

For sums and differences of vectors and matrices, the intrinsic array operations "+" and "–" are available. It is not necessary to have separate defined operations. A parsing rule in Fortran 90 states that the result of a defined operation involving two quantities has a lower precedence than any intrinsic operation. This explains the parentheses around the next-to-last line containing the sub-expression "`A .x. y`" found in the example. Users are advised to always include parentheses around array expressions that are mixed with defined operations, or whenever there is possible confusion without them. The next-to-last line of the example results in computing the residual associated with the solution, namely $r = b - Ay$. Ideally, this residual is zero when the system has a unique solution. It will be computed as a non-zero vector due to rounding errors and conditioning of the problem.

Matrix and Utility Functions

Several decompositions and functions required for numerical linear algebra follow. The convention of enclosing optional quantities in brackets, "[ ]" is used. The functions that use MPI for parallel execution of the box data type are marked in **`bold`**.

Defined Array Functions	Matrix Operation
`S=SVD(A [,U=U, V=V])`	$A = USV^T$

`E=EIG(A [[,B=B, D=D],`
`V=V, W=W])`
	$(AV = VE), AVD = BVE$
	$(AW = WE), AWD = BWE$

`R=CHOL(A)`	$A = R^T R$
`Q=ORTH(A [,R=R])`	$\left(A = QR\right), Q^T Q = I$
`U=UNIT(A)`	$\left[u_1, \ldots\right] = \left[a_1 / \|a_1\|, \ldots\right]$
`F=DET(A)`	$det(A) = determinant$
`K=RANK(A)`	$rank(A) = rank$
`P=NORM(A[,[type=]i])`	$p = \|A\|_1 = \max_j (\sum_{i=1}^{m} \lvert a_{ij} \rvert)$ $p = \|A\|_2 = s_1 =$ largest singular value $p = \|A\|_{\infty \leftrightarrow huge(1)} = \max_i (\sum_{j=1}^{n} \lvert a_{ij} \rvert)$
`C=COND(A)`	$s_1 / s_{rank(A)}$
`Z=EYE(N)`	$Z = I_N$
`A=DIAG(X)`	$A = diag\left(x_1, \ldots\right)$

Defined Array Functions	**Matrix Operation**
`X=DIAGONALS(A)`	$x = (a_{11}, \ldots)$

`Y=FFT (X,[WORK=W]);`
`X=IFFT(Y,[WORK=W])`
	Discrete Fourier Transform, Inverse

`Y=FFT_BOX (X,[WORK=W]);`
`X=IFFT_BOX(Y,[WORK=W])`
	Discrete Fourier Transform for Boxes, Inverse

`A=RAND(A)`	random numbers, $0 < A < 1$
`L=isNaN(A)`	test for `NaN`, *if (l) then…*

In certain functions, the optional arguments are inputs while other optional arguments are outputs. To illustrate the example of the box **`SVD`** function, a code is given that computes the singular value decomposition and the reconstruction of the random matrix box, A. Using the computed factors, $R = USV^T$. Mathematically $R = A$, but this will be true, only approximately, due to rounding errors. The value *units_of_error* = $\|A - R\|/(\|A\|\varepsilon)$, shows the merit of this approximation.

```
USE linear_operators

USE mpi_setup_int

   integer,parameter :: n=3, k=16

   real, dimension(n,n,k) :: A,U,V,R,S(n,k), units_of_error(k)

   MP_NPROCS=MP_SETUP()       ! Set up MPI.

   A=rand(A); S=SVD(A, U=U, V=V)

   R = U .x. diag(S) .xt. V; units_of_error =
      norm(A-R)/S(1,1:k)/epsilon(A)

   MP_NPROCS=MP_SETUP(‘Final’) ! Shut down MPI.

   end
```

Parallelism Using MPI

MPI REQUIRED

General Remarks

The central theme we use for the computing functions of the box data type is that of delivering results to a distinguished node of the machine. One of the design goals was to shield much of the complexity of distributed computing from the user.

The nodes are numbered by their “ranks.” Each node has *rank value* `MP_RANK`. There are `MP_NPROCS` nodes, so `MP_RANK = 0, 1,...,MP_NPROCS-1.` The root node has `MP_RANK = 0.` Most of the elementary MPI material is found in Gropp, Lusk, and Skjellum (1994) and Snir, Otto, Huss-Lederman, Walker, and Dongarra (1996). Although `Fortran Library` users are for the most part shielded from the complexity of MPI, it is desirable for some users to learn this important topic. Users should become familiar with any referenced MPI routines and the documentation of their usage. MPI routines are not discussed here, because that is best found in the above references.

The Fortran Library algorithm for allocating the racks of the box to the processors consists of creating a schedule for the processors, followed by communication and execution of this schedule. The efficiency may be improved by using the nodes according to a specific *priority order*. This order can reflect information such as a powerful machine on the network other than the user's work station, or even complex or transient network behavior. The Fortran Library allows users to define this order, including using a default. A setup function establishes an order based on timing matrix products of a size given by the user. Parallel Example 4 illustrates this usage.

Getting Started with Modules MPI_setup_int and MPI_node_int

The MPI_setup_int and MPI_node_int modules are part of the Fortran Library and not part of MPI itself. Following a call to the function MP_SETUP(), the module MPI_node_int will contain information about the number of processors, the rank of a processor, the communicator for Fortran Library, and the usage priority order of the node machines. Since MPI_node_int is used by MPI_setup_int, it is not necessary to explicitly use this module. If neither MP_SETUP() nor MPI_Init() is called, then the box data type will compute entirely on one node. No routine from MPI will be called.

```
MODULE MPI_NODE_INT
  INTEGER, ALLOCATABLE :: MPI_NODE_PRIORITY(:)
  INTEGER, SAVE :: MP_LIBRARY_WORLD = huge(1)
  LOGICAL, SAVE :: MPI_ROOT_WORKS = .TRUE.
  INTEGER, SAVE :: MP_RANK = 0, MP_NPROCS = 1
END MODULE
```

When the function MP_SETUP() is called with no arguments, the following events occur:

- If MPI has not been initialized, it is first initialized. This step uses the routines MPI_Initialized() and possibly MPI_Init(). Users who choose not to call MP_SETUP() must make the required initialization call before using any Fortran Library code that relies on MPI for its execution. If the user's code calls a Fortran Library function utilizing the box data type and MPI has not been initialized, then the computations are performed on the root node. The only MPI routine always called in this context is MPI_Initialized(). The name MP_SETUP is pushed onto the subprogram or call stack.

- If MP_LIBRARY_WORLD equals its initial value (=huge(1)) then MPI_COMM_WORLD, the default MPI communicator, is duplicated and becomes its handle. This uses the routine MPI_Comm_dup(). Users can change the handle of MP_LIBRARY_WORLD as required by their application code. Often this issue can be ignored.

- The integers MP_RANK and MP_NPROCS are respectively the node's rank and the number of nodes in the communicator, MP_LIBRARY_WORLD. Their values require the routines MPI_Comm_size() and MPI_Comm_rank(). The default values are important when MPI is not initialized and a box data type is computed. In this case the root node is the only node and it will do all the work. No calls to MPI communication routines are made when MP_NPROCS = 1 when computing the box data type functions. A program can temporarily assign this value to force box data type computation entirely at the root node. This is desirable for problems where using many nodes would be less efficient than using the root node exclusively.

- The array MPI_NODE_PRIORITY(:) is unallocated unless the user allocates it. The Fortran Library codes use this array for assigning tasks to processors, if it is allocated. If it is not allocated, the default priority of the nodes is (0,1,...,MP_NPROCS-1). Use of the function call MP_SETUP(N) allocates the array, as explained below. Once the array is allocated its size is MP_NPROCS. The contents of the array is a permutation of the integers 0,...,MP_NPROCS-1. Nodes appearing at the start of the list are used first for parallel computing. A node other than the root can avoid any computing, except receiving the schedule, by setting the value MPI_NODE_PRIORITY(I) < 0. This means that node |MPI_NODE_PRIORITY(I)| will be sent the task schedule but will not perform any significant work as part of box data type function evaluations.

- The LOGICAL flag MPI_ROOT_WORKS designates whether or not the root node participates in the major computation of the tasks. The root node communicates with the other nodes to complete the tasks but can be designated to do no other work. Since there may be only one processor, this flag has the default value .TRUE., assuring that one node exists to do work. When more than one processor is available users can consider assigning MPI_ROOT_WORKS=.FALSE. This is desirable when the alternate nodes have equal or greater computational resources compared with the root node. Example 4 illustrates this usage. A single problem is given a box data type, with one rack. The computing is done at the node, other than the root, with highest priority. This example requires more than one processor since the root does not work.

When the generic function MP_SETUP(N) is called, where N is a positive integer, a call to MP_SETUP() is first made, using no argument. Use just one of these calls to MP_SETUP(). This initializes the MPI system and the other parameters described above. The array MPI_NODE_PRIORITY(:) is allocated with size MP_NPROCS. Then DOUBLE PRECISION matrix products $C = AB,$ where A and B are N by N matrices, are computed at each node and the elapsed time is recorded. These elapsed times are sorted and the contents of MPI_NODE_PRIORITY(:) permuted in accordance with the shortest times yielding the highest priority. All the nodes in the communicator MP_LIBRARY_WORLD are timed. The array MPI_NODE_PRIORITY(:) is then broadcast from the root to the remaining nodes of MP_LIBRARY_WORLD using the routine MPI_Bcast(). Timing matrix products to define the node priority is relevant because the effort to compute C is comparable to that of many linear algebra computations of similar size. Users are free to define their own node priority and broadcast the array MPI_NODE_PRIORITY(:) to the alternate nodes in the communicator.

To print any IMSL Fortran Library error messages that have occurred at any node, and to finalize MPI, use the function call MP_SETUP('Final'). Case of the string 'Final' is not important. Any error messages pending will be discarded after printing on the root node. This is triggered by popping the name 'MP_SETUP' from the subprogram stack or returning to Level 1 in the stack. Users can obtain error messages by popping the stack to Level 1 and still continuing with MPI calls. This requires executing call e1pop ('MP_SETUP'). To continue on after summarizing errors execute call e1psh ('MP_SETUP'). More details about the error processor are found in Reference Material chapter of this manual.

Messages are printed by nodes from largest rank to smallest, which is the root node. Use of the routine MPI_Finalize() is made within MP_SETUP('Final'), which shuts down MPI. After MPI_Finalize() is called, the value of MP_NPROCS = 0. This flags that MPI has been initialized and terminated. It cannot be initialized again in the same program unit execution. No MPI routine is defined when MP_NPROCS has this value.

Using Processors

There are certain pitfalls to avoid when using Fortran Library and box data types as implemented with MPI. A fundamental requirement is to allow all processors to participate in parts of the program where their presence is needed for correctness. It is incorrect to have a program unit that restricts nodes from executing a block of code required when computing with the box data type. On the other hand it is appropriate to restrict computations with rank-2 arrays to the root node. This is not required, but the results for the alternate nodes are normally discarded. This will avoid gratuitous error messages that may appear at alternate nodes.

Observe that only the root has a correct result for a box data type function. Alternate nodes have the constant value one as the result. The reason for this is that during the computation of the functions, sub-problems are allocated to the alternate nodes by the root, but for only the root to utilize the result. If a user needs a value at the other nodes, then the root must send it to the nodes. This principle is illustrated in Parallel Example 3: Convergence information is computed at the root node and broadcast to the others. Without this step some nodes would not terminate the loop even when corrections at the root become small. This would cause the program to be incorrect.

Optional Data Changes

To reset tolerances for determining singularity and to allow for other data changes, non-allocated "hidden" variables are defined within the modules. These variables can be allocated first, then assigned values which result in the use of different tolerances or greater efficiency in the executable program. The non-allocated variables, whose scope is limited to the module, are hidden from the casual user. Default values or rules are applied if these arrays are not allocated. In more detail, the inverse matrix operator **".i."** applied to a square matrix first uses the *LU* factorization code `lin_sol_gen` and row pivoting. The default value for a small diagonal term is defined to be:

```
sqrt(epsilon(A))*sum(abs(A))/(n*n+1)
```

If the system is singular, a generalized matrix inverse is computed with the *QR* factorization code `lin_sol_lsq` using this same tolerance. Both row and column pivoting are used. If the system is singular, an error message will be printed and a Fortran 90 `STOP` is executed. Users may want to change this rule. This is illustrated by continuing and not printing the error message. The following is an additional source to accomplish this, for all following invocations of the operator **".i."**:

```
allocate(inverse_options(1))

inverse_options(1)=skip_error_processing

B=.i. A
```

There are additional self-documenting integer parameters, packaged in the module *linear_operators,* that allow users other choices, such as changing the value of the tolerance, as noted above. Included will be the ability to have the option apply for just the next invocation of the operator. Options are available that allow optional data to be passed to supporting Fortran 90 subroutines. This is illustrated with an example in `operator_ex36` in this chapter.

Operators: .x., .tx., .xt., .hx., .xh.

Computes matrix-vector and matrix-matrix products. The results are in a precision and data type that ascends to the most accurate or complex operand. The operators apply when one or both operands are rank-1, rank-2 or rank-3 arrays.

Required Operands

Each of these operators requires two operands. Mixing of intrinsic floating-point data types arrays is permitted. There is no distinction made between a rank-1 array, considered a slim matrix, and the transpose of this matrix. Defined operations have lower precedence than any intrinsic operation, so the liberal use of parentheses is suggested when mixing them.

Optional Variables, Reserved Names

These operators have neither packaged optional variables nor reserved names.

Modules

Use the appropriate one of the modules:

```
operation_x
operation_tx
operation_xt
operation_hx
operation_xh
or linear_operators
```

Examples

Compute the matrix times vector $y = Ax$: `y = A .x. x`

Compute the vector times matrix $y = x^T A$: `y = x .x.A; y = A .tx. x`

Compute the matrix expression $D = B - AC$: `D = B – (A .x. C)`

Operators: .t., .h.

Computes transpose and conjugate transpose of a matrix. The operation may be read *transpose or adjoint*, and the results are the mathematical objects in a precision and data type that matches the operand. The operators apply when the single operand is a rank-2 or rank-3 array.

Required Operand

Each of these operators requires a single operand. Since these are unary operations, they have *higher* Fortran 90 precedence than any other intrinsic unary array operation.

Optional Variables, Reserved Names

These operators have neither packaged optional variables nor reserved names.

Modules

Use the appropriate one of the modules:

```
operation_t
operation_h
or linear_operators
```

Examples

Compute the matrix times vector
$y = A^T x$: `y = .t.A .x. x; y = A .tx. x`

Compute the vector times matrix
$y = x^T A$: `y = x .x. A; y = A .tx. x`

Compute the matrix expression
$D = B - A^H C$: `D = B - (A .hx. C); D = B - (.h.A .x. C)`

Operator: .i.

Computes the inverse matrix, for square non-singular matrices, or the Moore-Penrose generalized inverse matrix for singular square matrices or rectangular matrices. The operation may be read *inverse or generalized inverse*, and the results are in a precision and data type that matches the operand. The operator can be applied to any rank-2 or rank-3 array.

Required Operand

This operator requires a single operand. Since this is a unary operation, it has *higher* Fortran 90 precedence than any other intrinsic array operation.

Optional Variables, Reserved Names

This operator uses the routines `lin_sol_gen` or `lin_sol_lsq` (See Chapter 1, "Linear Solvers" `lin_sol_gen` and `lin_sol_lsq).`

The option and derived type names are given in the following tables:

Option Names for `.i.`	**Option Value**
`use_lin_sol_gen_only`	1
`use_lin_sol_lsq_only`	2
`i_options_for_lin_sol_gen`	3
`i_options_for_lin_sol_lsq`	4
`skip_error_processing`	5

Derived Type	**Name of Unallocated Array**
`s_options`	`s_inv_options(:)`
`s_options`	`s_inv_iptions_once(:)`
`d_options`	`d_inv_options(:)`
`d_options`	`d_inv_options_once(:)`

Modules

Use the appropriate one of the modules:

`operation_i`

or `linear_operators`

Examples

Compute the matrix times vector

$y = A^{-1}x$: `y = .i.A .x. x ; y = A .ix. x`

Compute the vector times matrix

$y = x^{T}A^{-1}$: `y = x .x. .i.A; y = x .xi. A`

Compute the matrix expression

$D = B - A^{-1}C$: `D = B – (.i.A .x. C); D = B – (A .ix. C)`

Operators: .ix., .xi.

Computes the inverse matrix times a vector or matrix for square non-singular matrices or the corresponding Moore-Penrose generalized inverse matrix for singular square matrices or rectangular matrices. The operation may be read *generalized inverse times* or *times generalized inverse*. The results are in a precision and data type that matches the most accurate or complex operand.

Required Operand

This operator requires two operands. In the template for usage, `y = A .ix. b`, the first operand `A` can be rank-2 or rank-3. The second operand `b` can be rank-1, rank-2 or rank-3. For the alternate usage template, `y = b .xi. A`, the first operand `b` can be rank-1, rank-2 or rank-3. The second operand `A` can be rank-2 or rank-3.

Optional Variables, Reserved Names

This operator uses the routines `lin_sol_gen` or `lin_sol_lsq`
(See Chapter 1, “Linear Solvers”, `lin_sol_gen` and `lin_sol_lsq`).

The option and derived type names are given in the following tables:

Option Names for **`.ix., .xi.`**	Option Value
`use_lin_sol_gen_only`	1
`use_lin_sol_lsq_only`	2
`xi_, ix_options_for_lin_sol_gen`	3
`xi_, ix_options_for_lin_sol_lsq`	4
`skip_error_processing`	5

Derived Type	Name of Unallocated Array
`s_options`	`s_invx_options(:)`
`s_options`	`s_invx_options_once(:)`
`d_options`	`d_invx_options(:)`
`d_options`	`d_invx_options_once(:)`
`s_options`	`s_xinv_options(:)`
`s_options`	`s_xinv_options_once(:)`
`d_options`	`d_xinv_options(:)`
`d_options`	`d_xinv_options_once(:)`

Modules

Use the appropriate one of the modules:

```
operation_ix
operation_xi
or linear_operators
```

Examples

Compute the matrix times vector $y = A^{-1}x$: `y = A .ix. x`

Compute the vector times matrix $y = x^T A^{-1}$: `y = x .xi. A`

Compute the matrix expression $D = B - A^{-1}C$: `D = B - (A .ix. C)`

CHOL

Computes the Cholesky factorization of a positive-definite, symmetric or self-adjoint matrix, A. The factor is upper triangular, $R^T R = A$.

Required Argument

This function requires one argument. This argument must be a rank-2 or rank-3 array that contains a positive-definite, symmetric or self-adjoint matrix. For rank-3 arrays each rank-2 array, (for fixed third subscript), is a positive-definite, symmetric or self-adjoint matrix. In this case, the output is a rank-3 array of Cholesky factors for the individual problems.

Optional Variables, Reserved Names

This function uses `lin_sol_self` (See Chapter 1, "Linear Solvers," lin_sol_self), using the appropriate options to obtain the Cholesky factorization.

The option and derived type names are given in the following tables:

Option Name for `CHOL`	Option Value
`use_lin_sol_gen_only`	4
`use_lin_sol_lsq_only`	5

Derived Type	Name of Unallocated Array
`s_options`	`s_chol_options(:)`
`s_options`	`s_chol_options_once(:)`
`d_options`	`d_chol_options(:)`
`d_options`	`d_chol_options_once(:)`

Modules

Use the appropriate one of the modules:

```
chol_int
```

or `linear_operators`

Example

Compute the Cholesky factor of a positive-definite symmetric matrix:

```
B = A .tx. A; R = CHOL(B); B = R .tx. R
```

COND

Computes the condition number of a rectangular matrix, *A*. The condition number is the ratio of the largest and the smallest positive singular values,

$$s_1 / s_{rank(A)}$$

or `huge(A)`, whichever is smaller.

Required Argument

This function requires one argument. This argument must be a rank-2 or rank-3 array. For rank-3 arrays, each rank-2 array section, (for fixed third subscript), is a separate problem. In this case, the output is a rank-1 array of condition numbers for each problem.

Optional Variables, Reserved Names

This function uses `lin_sol_svd` (see Chapter 1, “Linear Solvers,” `lin_sol_svd`), to compute the singular values of *A*.

The option and derived type names are given in the following tables:

Option Name for COND	Option Value
`s_cond_set_small`	1
`s_cond_for_lin_sol_svd`	2
`d_cond_set_small`	1
`d_cond_for_lin_sol_svd`	2
`c_cond_set_small`	1
`c_cond_for_lin_sol_svd`	2
`z_cond_set_small`	1
`z_cond_for_lin_sol_svd`	2

Derived Type	Name of Unallocated Array
`s_options`	`s_cond_options(:)`
`s_options`	`s_cond_options_once(:)`
`d_options`	`d_cond_options(:)`
`d_options`	`d_cond_options_once(:)`

Modules

Use the appropriate one of the modules:

```
cond_int
or linear_operators
```

Example

Compute the condition number:

```
B = A .tx. A; c = COND(B); c = COND(A)**2
```

DET

Computes the determinant of a rectangular matrix, A. The evaluation is based on the QR decomposition,

$$QAP = \begin{bmatrix} R_{k \times k} & 0 \\ 0 & 0 \end{bmatrix}$$

and $k = rank(A)$. Thus $det(A) = s \times det(R)$ where $s = det(Q) \times det(P) = \pm 1$.

Required Argument

This function requires one argument. This argument must be a rank-2 or rank-3 array that contains a rectangular matrix. For rank-3 arrays, each rank-2 array (for fixed third subscript), is a separate matrix. In this case, the output is a rank-1 array of determinant values for each problem. Even well-conditioned matrices can have determinants with values that have very large or very tiny magnitudes. The values may overflow or underflow. For this class of problems, the use of the logarithmic representation of the determinant found in `lin_sol_gen` or `lin_sol_lsq` is required.

Optional Variables, Reserved Names

This function uses `lin_sol_lsq` (see Chapter 1, "Linear Solvers" `lin_sol_lsq`) to compute the *QR* decomposition of *A*, and the logarithmic value of `det`(*A*), which is exponentiated for the result.

The option and derived type names are given in the following tables:

Option Name for `DET`	Option Value
`s_det_for_lin_sol_lsq`	1
`d_det_for_lin_sol_lsq`	1
`c_det_for_lin_sol_lsq`	1
`z_det_for_lin_sol_lsq`	1

Derived Type	Name of Unallocated Array
`S_options`	`s_det_options(:)`
`S_options`	`s_det_options_once(:)`
`D_options`	`d_det_options(:)`
`D_options`	`d_det_options_once(:)`

Modules

Use the appropriate one of the modules:

```
det_int
or linear_operators
```

Example

Compute the determinant of a matrix and its inverse:

```
b = DET(A); c = DET(.i.A); b=1./c
```

DIAG

Constructs a square diagonal matrix from a rank-1 array or several diagonal matrices from a rank-2 array. The dimension of the matrix is the value of the size of the rank-1 array.

Required Argument

This function requires one argument, and the argument must be a rank-1 or rank-2 array. The output is a rank-2 or rank-3 array, respectively. The use of `DIAG` may be obviated by observing that the defined operations `C = diag(x) .x. A` or `D = B .x. diag(x)` are respectively the array operations `C = spread(x, DIM=1,NCOPIES=size(A,1))*A`, and `D = B*spread(x,DIM=2,NCOPIES=size(B,2)).` These array products are not as easy to read as the defined operations using `DIAG` and matrix multiply, but their use results in a more efficient code.

Optional Variables, Reserved Names

This function has neither packaged optional variables nor reserved names.

Modules

Use the appropriate module:

```
diag_int
or linear_operators
```

Example

Compute the singular value decomposition of a square matrix A:

```
S = SVD(A,U=U,V=V)
```

Then reconstruct $A = USV^T$:

```
A = U .x.diag(S) .xt. V
```

DIAGONALS

Extracts a rank-1 array whose values are the diagonal terms of a rank-2 array argument. The size of the array is the smaller of the two dimensions of the rank-2 array. When the argument is a rank-3 array, the result is a rank-2 array consisting of each separate set of diagonals.

Required Argument

This function requires one argument, and the argument must be a rank-2 or rank-3 array. The output is a rank-1 or rank-2 array, respectively.

Optional Variables, Reserved Names

This function has neither packaged optional variables nor reserved names.

Modules

Use the appropriate one of the modules:

```
diagonals_int
or linear_operators
```

Example

Compute the diagonals of the matrix product RR^T:

```
x = DIAGONALS(R .xt. R)
```

EIG

Computes the eigenvalue-eigenvector decomposition of an ordinary or generalized eigenvalue problem.

For the ordinary eigenvalue problem, $Ax = ex$, the optional input “`B=`” is not used. With the generalized problem, $Ax = eBx$, the matrix B is passed as the array in the right-side of “`B=`”. The optional output “`D=`” is an array required only for the generalized problem and then only when the matrix B is singular.

The array of real eigenvectors is an optional output for both the ordinary and the generalized problem. It is used as “`V=`” where the right-side array will contain the eigenvectors. If any eigenvectors are complex, the optional output “`W=`” must be present. In that case “`V=`” should not be used.

Required Argument

This function requires one argument, and the argument must be a square rank-2 array or a rank-3 array with square first rank-2 sections. The output is a rank-1 or rank-2 complex array of eigenvalues.

Optional Variables, Reserved Names

This function uses `lin_eig_self`, `lin_eig_gen`, and `lin_geig_gen`, to compute the decompositions. See Chapter 1, “Linear Solvers” `lin_eig_self`, `lin_eig_gen`, and `lin_geig_gen`.

The option and derived type names are given in the following tables:

Option Name for `EIG`	Option Value
`options_for_lin_eig_self`	1
`options_for_lin_eig_gen`	2
`options_for_lin_geig_gen`	3
`Skip_error_processing`	5

Derived Type	Name of Unallocated Array
s_options	s_eig_options(:)
s_options	s_eig_options_once(:)
d_options	d_eig_options(:)
d_options	d_eig_options_once(:)

Modules

Use the appropriate module:

```
eig_int
or linear_operators
```

Example

Compute the maximum magnitude eigenvalue of a square matrix *A*. (The values are sorted by `EIG()` to be non-increasing in magnitude).

```
E = EIG(A); max_magnitude = abs(E(1))
```

Compute the eigenexpansion of a square matrix *B*:

```
E = EIG(B, W = W); B = W .x. diag(E) .xi. W
```

EYE

Creates a rank-2 square array whose diagonals are all the value one. The off-diagonals all have value zero.

Required Argument

This function requires one integer argument, the dimension of the rank-2 array. The output array is of type and kind `REAL(KIND(1E0))`.

Optional Variables, Reserved Names

This function has neither packaged optional variables nor reserved names.

Modules

Use the appropriate module:

```
eye_int
or linear_operators
```

Example

Check the orthogonality of a set of `n` vectors, `Q`:

```
e = norm(EYE(n) – (Q .hx. Q))
```

FFT

The Discrete Fourier Transform of a complex sequence and its inverse transform.

Required Argument

The function requires one argument, x. If x is an assumed shape complex array of rank 1, 2 or 3, the result is the complex array of the same shape and rank consisting of the DFT.

Optional Variables, Reserved Names

The optional argument is "WORK=,"3 a COMPLEX array of the same precision as the data. For rank-1 transforms the size of WORK is n+15. To define this array for each problem, set WORK(1) = 0. Each additional rank adds the dimension of the transform plus 15. Using the optional argument WORK increases the efficiency of the transform. This function uses fast_dft, fast_2dft, and fast_3dft from Chapter 3.

The option and derived type names are given in the following tables:

Option Name for FFT	Option Value
options_for_fast_dft	1

Derived Type	Name of Unallocated Array
s_options	s_fft_options(:)
s_options	s_fft_options_once(:)
d_options	d_fft_options(:)
d_options	d_fft_options_once(:)

Modules

Use the appropriate module:

```
fft_int
```

or `linear_operators`

Example

Compute the DFT of a random complex array:

```
x=rand(x); y=fft(x)
```

FFT_BOX

The Discrete Fourier Transform of several complex or real sequences.

Required Argument

The function requires one argument, `x`. If `x` is an assumed shape complex array of rank 2, 3 or 4, the result is the complex array of the same shape and rank consisting of the DFT for each of the last rank's indices.

Optional Variables, Reserved Names

The optional argument is "`WORK=`," a `COMPLEX` array of the same precision as the data. For rank-1 transforms the size of `WORK` is n+15. To define this array for each problem, set `WORK(1) = 0`. Each additional rank adds the dimension of the transform plus 15. Using the optional argument `WORK` increases the efficiency of the transform. This function uses routines `fast_dft`, `fast_2dft`, and `fast_3dft` from this chapter.

The option and derived type names are given in the following tables:

Option Name for FFT	Option Value
options_for_fast_dft	1

Derived Type	Name of Unallocated Array
`S_options`	`s_fft_box_options(:)`
`S_options`	`s_fft_box_options_once(:)`
`D_options`	`d_fft_box_options(:)`
`D_options`	`d_fft_box_options_once(:)`

Modules

Use the appropriate module:

```
fft_box_int
or linear_operators
```

Example

Compute the DFT of a random complex array:

```
x=rand(x); y=fft_box(x)
```

IFFT

The inverse of the Discrete Fourier Transform of a complex sequence.

Required Argument

The function requires one argument, `x`. If `x` is an assumed shape complex array of rank 1, 2 or 3, the result is the complex array of the same shape and rank consisting of the inverse DFT.

Optional Variables, Reserved Names

The optional argument is "WORK=," a COMPLEX array of the same precision as the data. For rank-1 transforms the size of WORK is n+15. To define this array for each problem, set WORK(1) = 0. Each additional rank adds the dimension of the transform plus 15. Using the optional argument WORK increases the efficiency of the transform. This function uses routines fast_dft, fast_2dft, and fast_3dft from Chapter 3.

The option and derived type names are given in the following tables:

Option Name for IFFT	Option Value
options_for_fast_dft	1

Derived Type	Name of Unallocated Array
s_options	s_ifft_options(:)
s_options	S_ifft_options_once(:)
d_options	D_ifft_options(:)
d_options	D_ifft_options_once(:)

Modules

Use the appropriate module:

```
ifft_int
or linear_operators
```

Example

Computes the DFT of a random complex array and its inverse transform:

```
x=rand(x); y=fft(x); x=ifft(y)
```

IFFT_BOX

The inverse Discrete Fourier Transform of several complex or real sequences.

Required Argument

The function requires one argument, x. If x is an assumed shape complex array of rank 2, 3 or 4, the result is the complex array of the same shape and rank consisting of the inverse DFT.

Optional Variables, Reserved Names

The optional argument is "WORK=," a COMPLEX array of the same precision as the data. For rank-1 transforms the size of WORK is n+15. To define this array for each problem, set WORK(1) = 0. Each additional rank adds the dimension of the transform plus 15. Using the optional argument `WORK` increases the efficiency of the transform. This function uses routines `fast_dft`, `fast_2dft`, and `fast_3dft` from Chapter 3.

The option and derived type names are given in the following tables:

Option Name for IFFT	Option Value
`Options_for_fast_dft`	1

Derived Type	Name of Unallocated Array
`S_options`	`s_ifft_box_options(:)`
`S_options`	`s_ifft_box_options_once(:)`
`D_options`	`d_ifft_box_options(:)`
`D_options`	`d_ifft_box_options_once(:)`

Modules

Use the appropriate module:

```
ifft_box_int
or linear_operators
```

Example

Computes the inverse DFT of a random complex array:

```
x=rand(x); x=ifft_box(y)
```

isNaN

This is a generic logical function used to test scalars or arrays for occurrence of an IEEE 754 Standard format of floating point (ANSI/IEEE 1985) NaN, or not-a-number. Either *quiet* or *signaling* NaNs are detected without an exception occurring in the test itself. The individual array entries are each examined, with bit manipulation, until the first NaN is located. For non-IEEE formats, the bit pattern tested for single precision is `transfer(not(0),1)`. For double precision numbers `x`, the bit pattern tested is equivalent to assigning the integer array `i(1:2) = not(0)`, then testing this array with the bit pattern of the integer array `transfer(x,i)`. This function is likely to be required whenever there is the possibility that a subroutine blocked the output with NaNs in the presence of an error condition.

Required Arguments

The argument can be a scalar or array of rank-1, rank-2 or rank-3. The output value tests `.true.` only if there is at least one NaN in the scalar or array. The values can be any of the four intrinsic floating-point types.

Optional Variables, Reserved Names

This function has neither packaged optional variables nor reserved names.

Modules

Use one of the modules:

```
isNaN_int
```

```
or linear_operators
```

Example

If there is not a NaN in an array A it is used to solve a linear system:

```
if(.not. isNaN(A)) x = A .ix. b
```

NaN

Returns, as a scalar function, a value corresponding to the IEEE 754 Standard format of floating point (ANSI/IEEE 1985) for NaN. For other floating point formats a special pattern is returned that tests .true. using the function isNaN().

Required Arguments

X (Input)
Scalar value of the same type and precision as the desired result, NaN. This input value is used only to match the type of output.

Optional Arguments

There are no optional arguments for this routine.

Example: Blocking Output

Arrays are assigned all NaN values, using single and double-precision formats. These are tested using the logical function routine, isNaN.

```
      use isnan_int

      implicit none

! This is Example 1 for NaN.
      integer, parameter :: n=3
      real(kind(1e0)) A(n,n); real(kind(1d0)) B(n,n)
      real(kind(1e0)), external :: s_NaN
      real(kind(1d0)), external :: d_NaN

! Assign NaNs to both A and B:
      A = s_Nan(1e0); B = d_Nan(1d0)
```

```
! Check that NaNs are noted in both A and B:
      if (isNan(A) .and. isNan(B)) then


         write (*,*) 'Example 1 for NaN is correct.'
      end if

      end
```

Description

The bit pattern used for single precision is `transfer (not(0),1)`. For double precision, the bit pattern for single precision is replicated by assigning the temporary integer array `i(1:2) = not(0)`, and then using the double-precision bit pattern `transfer(i,x)` for the output value.

Fatal and Terminal Error Messages

This routine has no error messages.

NORM

Computes the norm of a rank-1 or rank-2 array. For rank-3 arrays, the norms of each rank-2 array, in dimension 3, are computed.

Required Arguments

The first argument must be an array of rank-1, rank-2, or rank-3. An optional, second position argument can be used that provides a choice between the norms

$$l_1, l_2, \text{ and } l_\infty$$

If this optional argument, with keyword " `type=`" is not present, the l_2 norm is computed. The l_1 and l_∞ norms are likely to be less expensive to compute than the l_2 norm. Use of the option number `?_reset_default_norm` will switch the default from the l_2 to the l_1 or l_∞ norms.

Optional Variables, Reserved Names

If the l_2 norm is required, this function uses `lin_sol_svd` (see Chapter 1, "Linear Solvers," `lin_sol_svd`), to compute the largest singular value of *A*. For the other norms, Fortran 90 intrinsics are used.

The option and derived type names are given in the following tables:

Option Name for NORM	Option Value
`s_norm_for_lin_sol_svd`	1
`s_reset_default_norm`	2
`d_norm_for_lin_sol_svd`	1

Option Name for NORM	Option Value
d_reset_default_norm	2
c_norm_for_lin_sol_svd	1
c_reset_default_norm	2
z_norm_for_lin_sol_svd	1
z_reset_default_norm	2

Derived Type	Name of Unallocated Array
s_options	s_norm_options(:)
s_options	s_norm_options_once(:)
d_options	d_norm_options(:)
d_options	d_norm_options_once(:)

Modules

Use the appropriate modules:

```
norm_int
or linear_operators
```

Example

Compute three norms of an array. (Both assignments of n_2 yield the same value).

```
A: n_1 = norm(A,1); n_2 = norm(A,type=2); n_2=norm(A);
n_inf = norm(A,huge(1))
```

ORTH

Orthogonalizes the columns of a rank-2 or rank-3 array. The decomposition $A = QR$ is computed using a forward and backward sweep of the Modified Gram-Schmidt algorithm.

Required Arguments

The first argument must be an array of rank-2 or rank-3. An optional argument can be used to obtain the upper-triangular or upper trapezoidal matrix R. If this optional argument, with keyword “R=”, is present, the decomposition is complete. The array output contains the matrix Q. If the first argument is rank-3, the output array and the optional argument are rank-3.

Optional Variables, Reserved Names

The option and derived type names are given in the following tables:

Option Name for `ORTH`	Option Value
`skip_error_processing`	5

Derived Type	Name of Unallocated Array
`s_options`	`s_orth_options(:)`
`s_options`	`s_orth_options_once(:)`
`d_options`	`d_orth_options(:)`
`d_options`	`d_orth_options_once(:)`

Modules

Use the appropriate one of the modules:

`orth_int`

or `linear_operators`

Example

Compute the scaled sample variances, v, of an $m \times n$ linear least squares system, (`m` > `n`), $Ax \cong b$ `: Q = ORTH(A,R=R); G=.i. R; x = G .x. (Q .hx. b); v=DIAGONALS(G .xh. G); v=v*sum((b-(A .x. x))**2)/(m–n)`

RAND

Computes a scalar, rank-1, rank-2 or rank-3 array of random numbers. Each component number is positive and strictly less than one in value.

Required Arguments

The argument must be a scalar, rank-1, rank-2, or rank-3 array of any intrinsic floating-point type. The output function value matches the required argument in type, kind and rank. For complex arguments, the output values will be real and imaginary parts with random values of the same type, kind, and rank.

Optional Variables, Reserved Names

This function uses `rand_gen` to obtain the number of values required by the argument. The values are then copied using the `RESHAPE` intrinsic.

Note: If any of the arrays `s_rand_options(:)`, `s_rand_options_once(:)`, `d_rand_options(:)`, or `d_rand_options_once(:)` are allocated, they are passed as arguments to `rand_gen` using the keyword "`iopt=`".

The option and derived type names are given in the following table:

Derived Type	Name of Unallocated Array
`S_options`	`s_rand_options(:)`
`S_options`	`s_rand_options_once(:)`
`D_options`	`d_rand_options(:)`
`D_options`	`d_rand_options_once(:)`

Modules

Use the appropriate modules:

```
rand_int
```

or `linear_operators`

Examples

Compute a random digit:

$1 \le i \le n$: `i=rand(1e0)*n+1`

Compute a random vector:

x : x=rand(x)

RANK

Computes the mathematical rank of a rank-2 or rank-3 array.

Required Arguments

The argument must be rank-2 or rank-3 array of any intrinsic floating-point type. The output function value is an integer with a value equal to the number of singular values that are greater than a tolerance. The default value for this tolerance is $\varepsilon^{1/2}s_1$, where ε is machine precision and s_1 is the largest singular value of the matrix.

Optional Variables, Reserved Names

This function uses `lin_sol_svd` to compute the singular values of the argument. The singular values are then compared with the value of the tolerance to compute the rank.

The option and derived type names are given in the following tables:

Option Name for **RANK**	Option Value
`S_rank_set_small`	1
`S_rank_for_lin_sol_svd`	2

Option Name for RANK	Option Value
D_rank_set_small	1
D_rank_for_lin_sol_svd	2
C_rank_set_small	1
C_rank_for_lin_sol_svd	2
Z_rank_set_small	1
Z_rank_for_lin_sol_svd	2

Derived Type	Name of Unallocated Array
S_options	s_rank_options(:)
S_options	s_rank_options_once(:)
D_options	d_rank_options(:)
d_options	d_rank_options_once(:)

Modules

Use the appropriate one of the modules:

```
rank_int
or linear_operators
```

Example

Compute the rank of an array of random numbers and then the rank of an array where each entry is the value one:

```
A=rand(A); k=rank(A); A=1; k=rank(A)
```

SVD

Computes the singular value decomposition of a rank-2 or rank-3 array, $A = USV^T$.

Required Arguments

The argument must be rank-2 or rank-3 array of any intrinsic floating-point type. The keyword arguments "U=" and "V=" are optional. The output array names used on the right-hand side must have sizes that are large enough to contain the right and left singular vectors, *U* and *V*.

Optional Variables, Reserved Names

This function uses one of the routines lin_svd and lin_sol_svd. If a complete decomposition is required, lin_svd is used. If singular values only, or singular values and one of the right and left singular vectors are required, then lin_sol_svd is called.

The option and derived type names are given in the following tables:

Option Name for SVD	**Option Value**
`options_for_lin_svd`	1
`options_for_lin_sol_svd`	2
`skip_error_processing`	5

Derived Type `s_options`	**Name of Unallocated Array** `s_svd_options(:)`
`s_options`	`s_svd_options_once(:)`
`d_options`	`d_svd_options(:)`
`d_options`	`d_svd_options_once(:)`

Modules

Use the appropriate module:

```
svd_int
```

or `linear_operators`

Example

Compute the singular value decomposition of a random square matrix:

```
A=rand(A); S=SVD(A,U=U,V=V); A=U .x. diag(S) .xt. V
```

UNIT

Normalizes the columns of a rank-2 or rank-3 array so each has Euclidean length of value one.

Required Arguments

The argument must be a rank-2 or rank-3 array of any intrinsic floating-point type. The output function value is an array of the same type and kind, where each column of each rank-2 principal section has Euclidean length of value one.

Optional Variables, Reserved Names

This function uses a rank-2 Euclidean length subroutine to compute the lengths of the nonzero columns, which are then normalized to have lengths of value one. The subroutine carefully avoids overflow or damaging underflow by rescaling the sums of squares as required. There are no reserved names.

Modules

Use the appropriate one of the modules:

`unit_int`

or `linear_operators`

Example

Normalizes a set of random vectors: `A=UNIT(RAND(A))`.

Overloaded =, /=, etc., for Derived Types

To assist users in writing compact and readable code, the IMSL Fortran Library provides overloaded assignment and logical operations for the derived types `s_options`, `d_options`, `s_error`, and `d_error`. Each of these derived types has an individual record consisting of an integer and a floating-point number. The components of the derived types, in all cases, are named `idummy` followed by `rdummy`. In many cases, the item referenced is the component `idummy`. This integer value can be used exactly as any integer by use of the `component selector` character (%). Thus, a program could assign a value and test after calling a routine:

```
s_epack(1)%idummy = 0
call lin_sol_gen(A,b,x,epack=s_epack)
if (s_epack(1)%idummy > 0) call error_post(s_epack)
```

Using the overloaded assignment and logical operations, this code fragment can be written in the more readable form:

```
s_epack(1) = 0
call lin_sol_gen(A,b,x,epack=s_epack)
if (s_epack(1) > 0) call error_post(s_epack)
```

Generally the assignments and logical operations refer only to component `idummy`. The assignment "`s_epack(1)=0`" is equivalent to "`s_epack(1)=s_error(0,0E0)`". Thus, the floating-point component `rdummy` is assigned the value `0E0`. The assignment statement "`I=s_epack(1)`", for `I` an integer type, is equivalent to "`I=s_epack(1)%idummy`". The value of component `rdummy` is ignored in this assignment. For the logical operators, a single element of any of the IMSL Fortran Library derived types can be in either the first or second operand.

Derived Type	Overloaded Assignments and Tests						
s_options	I=s_options(1);s_options(1)=I	= =	/=	<	<=	>	>=
s_options	I=d_options(1);d_options(1)=I	= =	/=	<	<=	>	>=
d_epack	I=s_epack(1);s_epack(1)=I	= =	/=	<	<=	>	>=
d_epack	I=d_epack(1);d_epack(1)=I	= =	/=	<	<=	>	>=

In the examples, `operator_ex01`, …, `_ex37`, the overloaded assignments and tests have been used whenever they improve the readability of the code.

Operator Examples

This section presents an equivalent implementation of the examples in “Linear Solvers, “Singular Value and Eigenvalue Decomposition,” and a single example from “Fourier Tranforms Chapters 1 and 2, and a single example from Chapter 3.” In all cases, the examples have been tested for correctness using equivalent mathematical criteria. On the other hand, these criteria are not identical to the corresponding examples in all cases. In Example 1 for `lin_sol_gen`, `err = maxval(abs(res))/sum(abs(A) + abs(b))`is computed. In the operator revision of this example, `operator_ex01`, `err = norm(b – (A .x. x))/(norm(A)*norm(x) + norm(b))` is computed.

Both formulas for `err` yield values that are about `epsilon(A)`. To be safe, the larger value `sqrt(epsilon(A))` is used as the tolerance.

The operator version of the examples are shorter and intended to be easier to read.

To match the corresponding examples in Chapters 1, 2, and 10 to those using the operators, consult the following table:

Chapters 1, 2 and 3 Examples	Corresponding Operators
`Lin_sol_gen_ex1,_ex2,_ex3,_ex4`	`operator_ex01,_ex02,_ex03,_ex04`
`Lin_sol_self_ex1,_ex2,_ex3,_ex4`	`operator_ex05,_ex06,_ex07,_ex08`
`Lin_sol_lsq_ex1,_ex2,_ex3,_ex4`	`operator_ex09,_ex10,_ex11,_ex12`
`Lin_sol_svd_ex1,_ex2,_ex3,_ex4`	`operator_ex13,_ex14,_ex15,_ex16`
`Lin_sol_tri_ex1,_ex2,_ex3,_ex4`	`operator_ex17,_ex18,_ex19,_ex20`
`Lin_svd_ex1,_ex2,_ex3,_ex4`	`operator_ex21,_ex22,_ex23,_ex24`
`Lin_eig_self_ex1,_ex2,_ex3,_ex4`	`operator_ex25,_ex26,_ex27,_ex28`
`Lin_eig_gen_ex1,_ex2,_ex3,_ex4`	`operator_ex29,_ex30,_ex31,_ex32`
`Lin_geig_gen_ex1,_ex2,_ex3,_ex4`	`operator_ex33,_ex34,_ex35,_ex36`
`fast_dft_ex4`	`operator_ex37`

Table A: Examples and Corresponding Operators

Operator_ex01

```
      use linear_operators
      implicit none

! This is Example 1 for LIN_SOL_GEN, with operators and functions.

      integer, parameter :: n=32
      real(kind(1e0)) :: one=1.0e0, err
      real(kind(1e0)), dimension(n,n) :: A, b, x

! Generate random matrices for A and b:
      A = rand(A); b=rand(b)

! Compute the solution matrix of Ax = b.
      x = A .ix. b
```

```
! Check the results.
      err = norm(b - (A .x. x))/(norm(A)*norm(x)+norm(b))
      if (err <= sqrt(epsilon(one))) &
         write (*,*) 'Example 1 for LIN_SOL_GEN (operators) is correct.'
      end
```

Operator_ex02

```
      use linear_operators
      implicit none

! This is Example 2 for LIN_SOL_GEN using operators and functions.

      integer, parameter :: n=32
      real(kind(1e0)) :: one=1e0, err, det_A, det_i
      real(kind(1e0)), dimension(n,n) :: A, inv

! Generate a random matrix.
      A = rand(A)
! Compute the matrix inverse and its determinant.
      inv = .i.A; det_A = det(A)
! Compute the determinant for the inverse matrix.
      det_i = det(inv)
! Check the quality of both left and right inverses.
      err = (norm(EYE(n)-(A .x. inv))+norm(EYE(n)-(inv.x.A)))/cond(A)
      if (err <= sqrt(epsilon(one)) .and. abs(det_A*det_i - one) <= &
                 sqrt(epsilon(one))) &
      write (*,*) 'Example 2 for LIN_SOL_GEN (operators) is correct.'
      end
```

Operator_ex03

```
      use linear_operators
      implicit none

! This is Example 3 for LIN_SOL_GEN using operators.
      integer, parameter :: n=32
      real(kind(1e0)) :: one=1e0, zero=0e0, A(n,n), b(n), x(n)
      real(kind(1e0)) change_new, change_old
      real(kind(1d0)) :: d_zero=0d0, c(n), d(n,n), y(n)

! Generate a random matrix and right-hand side.
      A = rand(A); b= rand(b)

! Save double precision copies of the matrix and right-hand side.
      D = A
      c = b
! Compute single precision inverse to compute the iterative refinement.
      A = .i. A

! Start solution at zero.  Update it to an accurate solution
! with each iteration.
      y = d_zero
      change_old = huge(one)

      iterative_refinement: do
! Compute the residual with higher accuracy than the data.
         b = c - (D .x. y)

! Compute the update in single precision.
         x = A .x. b
         y = x + y
         change_new = norm(x)

! Exit when changes are no longer decreasing.
         if (change_new >= change_old) exit iterative_refinement
         change_old = change_new
      end do iterative_refinement

      write (*,*) 'Example 3 for LIN_SOL_GEN (operators) is correct.'
      end
```

Operator_ex04

```
      use linear_operators

      implicit none

! This is Example 4 for LIN_SOL_GEN using operators.

      integer, parameter :: n=32, k=128
      integer i
      real(kind(1e0)), parameter :: one=1e0, t_max=1, delta_t=t_max/(k-1)
      real(kind(1e0)) err, A(n,n)
      real(kind(1e0)) t(k), y(n,k), y_prime(n,k)
      complex(kind(1e0)) x(n,n), z_0(n), y_0(n), d(n)

! Generate a random coefficient matrix.
      A = rand(A)

! Compute the eigenvalue-eigenvector decomposition
! of the system coefficient matrix.
      D = EIG(A, W=X)

! Generate a random initial value for the ODE system.
      y_0 = rand(y_0)

! Solve complex data system that transforms the initial
! values, X z_0=y_0.
      z_0 = X .ix. y_0

! The grid of points where a solution is computed:
      t = (/(i*delta_t,i=0,k-1)/)

! Compute y and y' at the values t(1:k).
! With the eigenvalue-eigenvector decomposition AX = XD, this
! is an evaluation of EXP(A t)y_0 = y(t).
      y = X .x. exp(spread(d,2,k)*spread(t,1,n))*spread(z_0,2,k)
```

```
! This is y', derived by differentiating y(t).
      y_prime  = X .x. spread(d,2,k)*exp(spread(d,2,k)*spread(t,1,n))* &
                       spread(z_0,2,k)

! Check results. Is  y' - Ay = 0?
      err = norm(y_prime-(A .x. y))/(norm(y_prime)+norm(A)*norm(y))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 4 for LIN_SOL_GEN (operators) is correct.'
      end if

      end
```

Operator_ex05

```
      use linear_operators
      implicit none

! This is Example 1 for LIN_SOL_SELF using operators and functions.
      integer, parameter :: m=64, n=32
      real(kind(1e0)) :: one=1.0e0, err
      real(kind(1e0)) A(n,n), b(n,n), C(m,n), d(m,n), x(n,n)

! Generate two rectangular random matrices.
      C = rand(C); d=rand(d)

! Form the normal equations for the rectangular system.
      A = C .tx. C; b = C .tx. d

! Compute the solution for Ax = b, A is symmetric.
      x = A .ix. b

! Check the results.
      err = norm(b - (A .x. x))/(norm(A)+norm(b))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for LIN_SOL_SELF (operators) is correct.'
      end if

      end
```

Operator_ex06

```
      use linear_operators

      implicit none

! This is Example 2 for LIN_SOL_SELF using operators and functions.

      integer, parameter :: m=64, n=32
      real(kind(1e0)) :: one=1e0, zero=0e0, err
      real(kind(1e0)) A(n,n), b(n), C(m,n), d(m), cov(n,n), x(n)

! Generate a random rectangular matrix and right-hand side.
      C = rand(C); d=rand(d)
```

```
! Form the normal equations for the rectangular system.
      A = C .tx. C; b = C .tx. d
      COV = .i. CHOL(A); COV = COV .xt. COV

! Compute the least-squares solution.
       x = C .ix. d

! Compare with solution obtained using the inverse matrix.
      err = norm(x - (COV .x. b))/norm(cov)

! Scale the inverse to obtain the sample covariance matrix.
      COV = sum((d - (C .x. x))**2)/(m-n) * COV
! Check the results.
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 2 for LIN_SOL_SELF (operators) is correct.'
      end if

      end
```

Operator_ex07

```
      use linear_operators

      implicit none

! This is Example 3 (using operators) for LIN_SOL_SELF.

      integer tries
      integer, parameter :: m=8, n=4, k=2
      integer ipivots(n+1)
      real(kind(1d0)) :: one=1.0d0, err
      real(kind(1d0)) a(n,n), b(n,1), c(m,n), x(n,1), &
             e(n), ATEMP(n,n)
      type(d_options) :: iopti(4)

! Generate a random rectangular matrix.
      C = rand(C)

! Generate a random right hand side for use in the inverse
! iteration.
      b = rand(b)

! Compute the positive definite matrix.
      A = C .tx. C; A = (A+.t.A)/2

! Obtain just the eigenvalues.
      E = EIG(A)

! Use packaged option to reset the value of a small diagonal.
      iopti(4) = 0
      iopti(1) = d_options(d_lin_sol_self_set_small,&
                 epsilon(one)*abs(E(1)))

! Use packaged option to save the factorization.
```

```
      iopti(2) = d_lin_sol_self_save_factors

! Suppress error messages and stopping due to singularity
! of the matrix, which is expected.
      iopti(3) = d_lin_sol_self_no_sing_mess

      ATEMP = A

! Compute A-eigenvalue*I as the coefficient matrix.
! Use eigenvalue number k.
      A = A - e(k)*EYE(n)

      do tries=1,2
         call lin_sol_self(A, b, x, &
                     pivots=ipivots, iopt=iopti)
! When code is re-entered, the already computed factorization
! is used.
         iopti(4) = d_lin_sol_self_solve_A

! Reset right-hand side in the direction of the eigenvector.
         B = UNIT(x)
      end do

! Normalize the eigenvector.
      x = UNIT(x)

! Check the results.
      b=ATEMP .x. x
      err =  dot_product(x(1:n,1), b(1:n,1)) - e(k)

! If any result is not accurate, quit with no printing.
      if (abs(err) <= sqrt(epsilon(one))*E(1)) then
        write (*,*) 'Example 3 for LIN_SOL_SELF (operators) is correct.'
      end if

      end
```

Operator_ex08

```
      use linear_operators
      implicit none

! This is Example 4 for LIN_SOL_SELF using operators and functions.

      integer, parameter :: m=8, n=4
      real(kind(1e0)) :: one=1e0, zero=0e0
      real(kind(1d0)) :: d_zero=0d0
      integer ipivots((n+m)+1)
      real(kind(1e0)) A(m,n), b(m,1), F(n+m,n+m),&
            g(n+m,1), h(n+m,1)
      real(kind(1e0)) change_new, change_old
      real(kind(1d0)) c(m,1), D(m,n), y(n+m,1)
      type(s_options) ::  iopti(2)

! Generate a random matrix and right-hand side.
```

```
      A = rand(A); b = rand(b)

! Save double precision copies of the matrix and right hand side.
      D = A; c = b

! Fill in augmented matrix for accurately solving the least-squares
! problem using iterative refinement.
      F = zero; F(1:m,1:m)=EYE(m)
      F(1:m,m+1:) = A; F(m+1:,1:m) = .t. A

! Start solution at zero.
      y = d_zero
      change_old = huge(one)

! Use packaged option to save the factorization.
      iopti(1) = s_lin_sol_self_save_factors
      iopti(2) = 0

      iterative_refinement: do
         g(1:m,1) = c(1:m,1) - y(1:m,1) - (D .x. y(m+1:m+n,1))
         g(m+1:m+n,1) = - D .tx. y(1:m,1)
         call lin_sol_self(F, g, h, &
                   pivots=ipivots, iopt=iopti)
         y = h + y
         change_new = norm(h)

! Exit when changes are no longer decreasing.
         if (change_new >= change_old)&
                    exit iterative_refinement
         change_old = change_new

! Use option to re-enter code with factorization saved; solve only.
         iopti(2) = s_lin_sol_self_solve_A
      end do iterative_refinement
      write (*,*) 'Example 4 for LIN_SOL_SELF (operators) is correct.'
      end
```

Operator_ex09

```
      use linear_operators
      use Numerical_Libraries
      implicit none

! This is Example 1 for LIN_SOL_LSQ using operators and functions.

      integer i
      integer, parameter :: m=128, n=8
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)) A(m,0:n), c(0:n), pi_over_2, x(m), y(m), &
              u(m), v(m), w(m), delta_x
       CHARACTER(2) :: PI(1)

! Generate a random grid of points and transform
! to the interval -1,1.
      x = rand(x); x = x*2 - one
```

```
! Get the constant 'PI/2' from IMSL Numerical Libraries.
      PI='pi'; pi_over_2 = DCONST(PI)/2

! Generate function data on the grid.
      y = exp(x) + cos(pi_over_2*x)

! Fill in the least-squares matrix for the Chebyshev polynomials.
      A(:,0) = one; A(:,1) = x

      do i=2, n
         A(:,i) = 2*x*A(:,i-1) - A(:,i-2)
      end do

! Solve for the series coefficients.
      c = A .ix. y

! Generate an equally spaced grid on the interval.
      delta_x = 2/real(m-1,kind(one))
      x = (/(-one + i*delta_x,i=0,m-1)/)

! Evaluate residuals using backward recurrence formulas.
      u = zero; v = zero
      do i=n, 0, -1
         w = 2*x*u - v + c(i)
         v = u
         u = w
      end do

! Compute residuals at the grid:
      y = exp(x) + cos(pi_over_2*x) - (u-x*v)

! Check that n+1 sign changes in the residual curve occur.
! (This test will fail when n is larger.)
      x = one
      x = sign(x,y)

      if (count(x(1:m-1) /= x(2:m)) >= n+1) then
         write (*,*) 'Example 1 for LIN_SOL_LSQ (operators) is correct.'
      end if

      end
```

Operator_ex10

```
      use linear_operators
      implicit none

! This is Example 2 for LIN_SOL_LSQ using operators and functions.

      integer i
      integer, parameter :: m=128, n=8
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)) A(m,0:n), c(0:n), pi_over_2, x(m), y(m), &
```

```
             u(m), v(m), w(m), delta_x, inv(0:n, m)
      real(kind(1d0)), external :: DCONST

! Generate an array of equally spaced points on the interval -1,1.
      delta_x = 2/real(m-1,kind(one))
      x = (/(-one + i*delta_x,i=0,m-1)/)

! Get the constant 'PI/2' from IMSL Numerical Libraries.
      pi_over_2 = DCONST('PI')/2

! Compute data values on the grid.
      y = exp(x) + cos(pi_over_2*x)

! Fill in the least-squares matrix for the Chebyshev polynomials.
      A(:,0) = one
      A(:,1) = x

      do i=2, n
         A(:,i) = 2*x*A(:,i-1) - A(:,i-2)
      end do

! Compute the generalized inverse of the least-squares matrix.
! Compute the series coefficients using the generalized inverse
! as 'smoothing formulas.'
      inv = .i. A; c = inv .x. y

! Evaluate residuals using backward recurrence formulas.

      u = zero
      v = zero
      do i=n, 0, -1
         w = 2*x*u - v + c(i)
         v = u
         u = w
      end do

! Compute residuals at the grid:
      y = exp(x) + cos(pi_over_2*x) - (u-x*v)

! Check that n+2 sign changes in the residual curve occur.
! (This test will fail when n is larger.)

      x = one; x = sign(x,y)

      if (count(x(1:m-1) /= x(2:m)) == n+2) then
         write (*,*) 'Example 2 for LIN_SOL_LSQ (operators) is correct.'
      end if

      end
```

Operator_ex11

```
      use operation_ix
      use operation_tx
      use operation_x
      use rand_int
      use norm_int
      implicit none

! This is Example 3 for LIN_SOL_LSQ using operators and functions.
      integer i, j
      integer, parameter :: m=128, n=32, k=2, n_eval=16
      real(kind(1d0)), parameter :: one=1d0, delta_sqr=1d0
      real(kind(1d0)) A(m,n), b(m), c(n), p(k,m), q(k,n), &
              res(n_eval,n_eval), w(n_eval), delta

! Generate a random set of data and center points in k=2 space.
      p = rand(p); q=rand(q)

! Compute the coefficient matrix for the least-squares system.
      A = sqrt(sum((spread(p,3,n) - spread(q,2,m))**2,dim=1) + delta_sqr)

! Compute the right-hand side of function values.
      b = exp(-sum(p**2,dim=1))

! Compute the least-squares solution.  An error message due
! to rank deficiency is ignored with the flags:

      allocate (d_invx_options(1))
      d_invx_options(1)=skip_error_processing
      c = A .ix. b

! Check the results.
      if (norm(A .tx. (b - (A .x. c)))/(norm(A)+norm(c)) &
          <= sqrt(epsilon(one))) then
         write (*,*) 'Example 3 for LIN_SOL_LSQ (operators) is correct.'
      end if

! Evaluate residuals, known function - approximation at a square
! grid of points.  (This evaluation is only for k=2.)

      delta = one/real(n_eval-1,kind(one))
      w = (/(i*delta,i=0,n_eval-1)/)

      res = exp(-(spread(w,1,n_eval)**2 + spread(w,2,n_eval)**2))
      do j=1, n
         res = res - c(j)*sqrt((spread(w,1,n_eval) - q(1,j))**2 + &
                    (spread(w,2,n_eval) - q(2,j))**2 + delta_sqr)
      end do
! Unload option type for good housekeeping.
      deallocate (d_invx_options)
      end
```

Operator_ex12

```
      use linear_operators
      implicit none

! This is Example 4 for LIN_SOL_LSQ using operators and functions.
```

```
      integer, parameter :: m=64, n=32
      real(kind(1e0)) :: one=1e0, A(m+1,n), b(m+1), x(n)

! Generate a random matrix and right-hand side.
      A=rand(A); b = rand(b)

! Heavily weight desired constraint.  All variables sum to one.
      A(m+1,:) = one/sqrt(epsilon(one))
      b(m+1)   = one/sqrt(epsilon(one))

! Compute the least-squares solution with this heavy weight.
      x = A .ix. b

! Check the constraint.
      if (abs(sum(x) - one)/norm(x) <= sqrt(epsilon(one))) then
         write (*,*) 'Example 4 for LIN_SOL_LSQ (operators) is correct.'
      end if

      end
```

Operator_ex13

```
      use linear_operators
      implicit none

! This is Example 1 for LIN_SOL_SVD using operators and functions.
      integer, parameter :: m=128, n=32
      real(kind(1d0)) :: one=1d0, err
      real(kind(1d0)) A(m,n), b(m), x(n), U(m,m), V(n,n), S(n), g(m)

! Generate a random matrix and right-hand side.
      A = rand(A); b = rand(b)

! Compute the least-squares solution matrix of Ax=b.
      S = SVD(A, U = U, V = V)
      g = U .tx. b; x = V .x. diag(one/S) .x. g(1:n)

! Check the results.
      err = norm(A .tx. (b - (A .x. x)))/(norm(A)+norm(x))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for LIN_SOL_SVD (operators) is correct.'
      end if

      end
```

Operator_ex14

```
      use linear_operators
      implicit none

! This is Example 2 for LIN_SOL_SVD using operators and functions.
      integer, parameter :: n=32
      real(kind(1d0)) :: one=1d0, zero=0d0
      real(kind(1d0)) A(n,n), P(n,n), Q(n,n), &
```

```
           S_D(n), U_D(n,n), V_D(n,n)

! Generate a random matrix.
      A = rand(A)

! Compute the singular value decomposition.
      S_D = SVD(A, U=U_D, V=V_D)

! Compute the (left) orthogonal factor.
      P = U_D .xt. V_D

! Compute the (right) self-adjoint factor.
      Q = V_D .x. diag(S_D) .xt. V_D

! Check the results.
      if (norm( EYE(n) - (P .xt. P)) &
               <= sqrt(epsilon(one))) then
         if (norm(A - (P .x. Q))/norm(A) &
               <= sqrt(epsilon(one))) then
            write (*,*) 'Example 2 for LIN_SOL_SVD (operators) is correct.'
         end if
      end if
      end
```

Operator_ex15

```
      use linear_operators

      implicit none

! This is Example 3 for LIN_SOL_SVD.
      integer i, j, k
      integer, parameter :: n=32
      real(kind(1e0)), parameter :: half=0.5e0, one=1e0, zero=0e0
      real(kind(1e0)), dimension(n,n) :: A, S(n), U, V, C

! Fill in value one for points inside the circle,
! zero on the outside.
      A = zero
      DO i=1, n
         DO j=1, n
            if ((i-n/2)**2 + (j-n/2)**2 <= (n/4)**2) A(i,j) = one
         END DO
      END DO

! Compute the singular value decomposition.
      S = SVD(A, U=U, V=V)

! How many terms, to the nearest integer, match the circle?
      k = count(S > half)
      C = U(:,1:k) .x. diag(S(1:k)) .xt. V(:,1:k)
```

```
       if (count(int(C-A) /= 0) == 0) then
         write (*,*) 'Example 3 for LIN_SOL_SVD (operators) is correct.'
      end if

      end
```

Operator_ex16

```
      use linear_operators

      implicit none

! This is Example 4 (operators) for LIN_SOL_SVD.

      integer i, j, k
      integer, parameter :: m=64, n=16
      real(kind(1e0)), parameter :: one=1e0, zero=0e0
      real(kind(1e0)) :: g(m), s(m), t(n+1), a(m,n), f(n), U_S(m,m), &
              V_S(n,n), S_S(n)
      real(kind(1e0)) :: delta_g, delta_t, rms, oldrms

! Compute collocation equations to solve.
      delta_g = one/real(m+1,kind(one))
      g = (/(i*delta_g,i=1,m)/)

! Compute equally spaced quadrature points.
      delta_t =one/real(n,kind(one))
      t=(/((j-1)*delta_t,j=1,n+1)/)

! Compute collocation points with an array form of
! Newton's method.
      s=m
      SOLVE_EQUATIONS: do
        s=s-(exp(-s)-(one-s*g))/(g-exp(-s))
        if (sum(abs((one-exp(-s))/s - g)) <= &
            epsilon(one)*sum(g))exit SOLVE_EQUATIONS
      end do SOLVE_EQUATIONS

! Evaluate the integrals over the quadrature points.
      A = (exp(-spread(t(1:n),1,m)  *spread(s,2,n)) &
        -  exp(-spread(t(2:n+1),1,m)*spread(s,2,n))) / &
           spread(s,2,n)

! Compute the singular value decomposition.
      S_S = SVD(A, U=U_S, V=V_S)

! Singular values, larger than epsilon, determine
! the rank, k.
      k = count(S_S > epsilon(one))

! Compute U_S**T times right-hand side, g.
      g = U_S .tx. g

! Use the minimum number of singular values that give a good
! approximation to f(t) = 1.
```

```
      oldrms = huge(one)
      do i=1,k
         f = V_S(:,1:i) .x. (g(1:i)/S_S(1:i))
         rms = sum((f-one)**2)/n
         if (rms > oldrms) exit
         oldrms = rms
      end do

      write (*,"( ' Using this number of singular values, ', &
          &i4 / ' the approximate R.M.S. error is ', 1pe12.4)") &
      i-1, oldrms

      if (sqrt(oldrms) <= delta_t**2) then
         write (*,*) 'Example 4 for LIN_SOL_SVD (operators) is correct.'
      end if

      end
```

Operator_ex17

```
      use linear_operators
      use lin_sol_tri_int

      implicit none
! This is Example 1 (using operators) for LIN_SOL_TRI.
integer, parameter :: n=128
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)) err
      real(kind(1d0)), dimension(2*n,n) :: d, b, c, x, y, t(n)
      type(d_error) :: d_lin_sol_tri_epack(08) = d_error(0,zero)

! Generate the upper, main, and lower diagonals of the
! n matrices A_i.  For each system a random vector x is used
! to construct the right-hand side, Ax = y.  The lower part
! of each array remains zero as a result.

      c = zero; d=zero; b=zero; x=zero
            c(1:n,:)=rand(c(1:n,:)); d(1:n,:)=rand(d(1:n,:))
            b(1:n,:)=rand(b(1:n,:)); x(1:n,:)=rand(x(1:n,:))

! Add scalars to the main diagonal of each system so that
! all systems are positive definite.
      t = sum(c+d+b,DIM=1)
      d(1:n,1:n) = d(1:n,1:n) + spread(t,DIM=1,NCOPIES=n)

! Set Ax = y.  The vector x generates y.  Note the use
! of EOSHIFT and array operations to compute the matrix
! product, n distinct copies, as one array operation.

     y(1:n,1:n)=d(1:n,1:n)*x(1:n,1:n) + &
                c(1:n,1:n)*EOSHIFT(x(1:n,1:n),SHIFT=+1,DIM=1) + &
                b(1:n,1:n)*EOSHIFT(x(1:n,1:n),SHIFT=-1,DIM=1)

! Compute the solution returned in y.  (The input values of c,
! d, b, and y are overwritten by lin_sol_tri.)  Check for any
! errors.  This is not recessary but illustrates control
! returning to the calling program unit.
      call lin_sol_tri (c, d, b, y, &
           epack=d_lin_sol_tri_epack)
      call error_post(d_lin_sol_tri_epack)

! Check the size of the residuals, y-x.  They should be small,
! relative to the size of values in x.

      err = norm(x(1:n,1:n) - y(1:n,1:n),1)/norm(x(1:n,1:n),1)
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for LIN_SOL_TRI (operators) is correct.'
      end if

      end
```

Operator_ex18

```
      use linear_operators
      use lin_sol_tri_int

      implicit none

! This is Example 2 (using operators) for LIN_SOL_TRI.
      integer nopt
      integer, parameter :: n=128
      real(kind(1e0)), parameter :: s_one=1e0, s_zero=0e0
      real(kind(1d0)), parameter :: d_one=1d0, d_zero=0d0
      real(kind(1e0)), dimension(2*n,n) :: d, b, c, x, y
      real(kind(1e0)) change_new, change_old, err
      type(s_options) :: iopt(2) = s_options(0,s_zero)
      real(kind(1d0)), dimension(n,n) :: d_save, b_save, c_save, &
              x_save, y_save, x_sol
      logical solve_only

      c = s_zero; d=s_zero; b=s_zero; x=s_zero

! Generate the upper, main, and lower diagonals of the
! matrices A.  A random vector x is used to construct the
! right-hand sides: y=A*x.
      c(1:n,:)=rand(c(1:n,:)); d(1:n,:)=rand(d(1:n,:))
      d(1:n,:)=rand(c(1:n,:)); x(1:n,:)=rand(d(1:n,:))

! Save double precision copies of the diagonals and the
! right-hand side.
      c_save = c(1:n,1:n); d_save = d(1:n,1:n)
      b_save = b(1:n,1:n); x_save = x(1:n,1:n)
      y_save(1:n,1:n) = d(1:n,1:n)*x_save + &
                c(1:n,1:n)*EOSHIFT(x_save,SHIFT=+1,DIM=1) + &
                b(1:n,1:n)*EOSHIFT(x_save,SHIFT=-1,DIM=1)


! Iterative refinement loop.
      factorization_choice:  do nopt=0, 1
```

```
! Set the logical to flag the first time through.

         solve_only = .false.
         x_sol = d_zero
         change_old = huge(s_one)

         iterative_refinement:  do

! This flag causes a copy of data to be moved to work arrays
! and a factorization and solve step to be performed.
            if (.not. solve_only) then
               c(1:n,1:n)=c_save; d(1:n,1:n)=d_save
               b(1:n,1:n)=b_save
            end if

! Compute current residuals, y - A*x, using current x.
            y(1:n,1:n) = -y_save + &
             d_save*x_sol + &
             c_save*EOSHIFT(x_sol,SHIFT=+1,DIM=1) + &
             b_save*EOSHIFT(x_sol,SHIFT=-1,DIM=1)

            call lin_sol_tri (c, d, b, y, iopt=iopt)

            x_sol = x_sol + y(1:n,1:n)

            change_new = sum(abs(y(1:n,1:n)))

! If size of change is not decreasing, stop the iteration.
            if (change_new >= change_old) exit iterative_refinement

            change_old = change_new
            iopt(nopt+1) = s_lin_sol_tri_solve_only
            solve_only = .true.

         end do iterative_refinement

! Use Gaussian Elimination if Cyclic Reduction did not get an
! accurate solution.
! It is an exceptional event when Gaussian Elimination is required.
         if (norm(x_sol - x_save,1) / norm(x_save,1) &
           <= sqrt(epsilon(d_one))) exit factorization_choice

         iopt(nopt+1) = s_lin_sol_tri_use_Gauss_elim

      end do factorization_choice

! Check on accuracy of solution.

      err = norm(x(1:n,1:n)- x_save,1)/norm(x_save,1)
      if (err <= sqrt(epsilon(d_one))) then
         write (*,*) 'Example 2 for LIN_SOL_TRI (operators) is correct.'
      end if

      end
```

Operator_ex19

```
      use linear_operators
      use lin_sol_tri_int
      use rand_int
      use Numerical_Libraries

      implicit none

! This is Example 3 (using operators) for LIN_SOL_TRI.

      integer i, nopt
      integer, parameter :: n=128, k=n/4, ncoda=1, lda=2
      real(kind(1e0)), parameter :: s_one=1e0, s_zero=0e0
      real(kind(1e0)) A(lda,n), EVAL(k)
      type(s_options) :: iopt(2)
      real(kind(1e0)) d(n), b(n), d_t(2*n,k), c_t(2*n,k), perf_ratio, &
           b_t(2*n,k), y_t(2*n,k), eval_t(k), res(n,k)
      logical small

! This flag is used to get the k largest eigenvalues.
      small = .false.

! Generate the main diagonal and the co-diagonal of the
! tridiagonal matrix.
      b=rand(b); d=rand(d)
      A(1,1:)=b; A(2,1:)=d

! Use Numerical Libraries routine for the calculation of k
! largest eigenvalues.
      CALL EVASB (N, K, A, LDA, NCODA, SMALL, EVAL)
      EVAL_T = EVAL

! Use Fortran Librarytridiagonal solver for inverse iteration
! calculation of eigenvectors.
      factorization_choice:  do nopt=0,1

! Create k tridiagonal problems, one for each inverse
! iteration system.
         b_t(1:n,1:k) = spread(b,DIM=2,NCOPIES=k)
         c_t(1:n,1:k) = EOSHIFT(b_t(1:n,1:k),SHIFT=1,DIM=1)
         d_t(1:n,1:k) = spread(d,DIM=2,NCOPIES=k) - &
                        spread(EVAL_T,DIM=1,NCOPIES=n)

! Start the right-hand side at random values, scaled downward
! to account for the expected 'blowup' in the solution.
         y_t=rand(y_t)

! Do two iterations for the eigenvectors.
         do i=1, 2
            y_t(1:n,1:k) = y_t(1:n,1:k)*epsilon(s_one)
            call lin_sol_tri(c_t, d_t, b_t, y_t, &
                        iopt=iopt)
            iopt(nopt+1) = s_lin_sol_tri_solve_only
         end do
```

```
! Orthogonalize the eigenvectors.  (This is the most
! intensive part of the computing.)
         y_t(1:n,1:k) = ORTH(y_t(1:n,1:k))


! See if the performance ratio is smaller than the value one.
! If it is not the code will re-solve the systems using Gaussian
! Elimination.  This is an exceptional event.  It is a necessary
! complication for achieving reliable results.

         res(1:n,1:k) = spread(d,DIM=2,NCOPIES=k)*y_t(1:n,1:k) + &
          spread(b,DIM=2,NCOPIES=k)* &
          EOSHIFT(y_t(1:n,1:k),SHIFT=-1,DIM=1) + &
          EOSHIFT(spread(b,DIM=2,NCOPIES=k)*y_t(1:n,1:k),SHIFT=1) &
            -   y_t(1:n,1:k)*spread(EVAL_T(1:k),DIM=1,NCOPIES=n)

! If the factorization method is Cyclic Reduction and perf_ratio is
! larger than one, re-solve using Gaussian Elimination.  If the
! method is already Gaussian Elimination, the loop exits
! and perf_ratio is checked at the end.
         perf_ratio = norm(res(1:n,1:k),1) / &
                      norm(EVAL_T(1:k),1) / &
                         epsilon(s_one) / (5*n)
         if (perf_ratio <= s_one) exit factorization_choice
         iopt(nopt+1) = s_lin_sol_tri_use_Gauss_elim

      end do factorization_choice

      if (perf_ratio <= s_one) then
         write (*,*) 'Example 3 for LIN_SOL_TRI (operators) is correct.'
      end if

      end
```

Operator_ex20

```
      use lin_sol_tri_int
      use Numerical_Libraries

      implicit none

! This is Example 4 (using operators) for LIN_SOL_TRI.

      integer, parameter :: n=1000, ichap=5, iget=1, iput=2, &
         inum=6, irnum=7
      real(kind(1e0)), parameter :: zero=0e0, one = 1e0
      integer    i, ido, in(50), inr(20), iopt(6), ival(7), &
                iwk(35+n)
      real(kind(1e0))      hx, pi_value, t, u_0, u_1, atol, rtol, sval(2), &
                tend, wk(41+11*n), y(n), ypr(n), a_diag(n), &
                a_off(n), r_diag(n), r_off(n), t_y(n), t_ypr(n), &
                t_g(n), t_diag(2*n,1), t_upper(2*n,1), &
                t_lower(2*n,1), t_sol(2*n,1)
      type(s_options) :: iopti(1)=s_options(0,zero)
```

```
! Define initial data.
      t = 0e0; u_0 = one
      u_1 = 0.5; tend = one

! Initial values for the variational equation.
      y = -one; ypr= zero
      pi_value = const((/'pi'/))
      hx = pi_value/(n+1)

      a_diag = 2*hx/3
      a_off  = hx/6
      r_diag = -2/hx
      r_off  = 1/hx

! Get integer and floating point option numbers.
      iopt(1) = inum
      call iumag ('math', ichap, iget, 1, iopt, in)
      iopt(1) = irnum
      call iumag ('math', ichap, iget, 1, iopt, inr)

! Set for reverse communication evaluation of the DAE.
      iopt(1) = in(26)
      ival(1) = 0
! Set for use of explicit partial derivatives.
      iopt(2) = in(5)
      ival(2) = 1
! Set for reverse communication evaluation of partials.
      iopt(3) = in(29)
      ival(3) = 0
! Set for reverse communication solution of linear equations.
      iopt(4) = in(31)
      ival(4) = 0
! Storage for the partial derivative array are not allocated or
! required in the integrator.
      iopt(5) = in(34)
      ival(5) = 1
! Set the sizes of iwk, wk for internal checking.
      iopt(6) = in(35)
      ival(6) = 35 + n
      ival(7) = 41 + 11*n
! Set integer options:
      call iumag ('math', ichap, iput, 6, iopt, ival)
! Reset tolerances for integrator:
      atol = 1e-3; rtol= 1e-3
      sval(1) = atol; sval(2) = rtol
      iopt(1) = inr(5)
! Set floating point options:
      call sumag ('math', ichap, iput, 1, iopt, sval)
! Integrate ODE/DAE.  Use dummy external names for g(y,y')
! and partials: DGSPG, DJSPG.
      ido = 1
      Integration_Loop: do

          call d2spg (n, t, tend, ido, y, ypr, dgspg, djspg, iwk, wk)
```

```
! Find where g(y,y') goes.  (It only goes in one place here, but can
! vary where divided differences are used for partial derivatives.)
          iopt(1) = in(27)
          call iumag ('math', ichap, iget, 1, iopt, ival)
! Direct user response:
        select case(ido)

        case(1,4)
! This should not occur.
          write (*,*) ' Unexpected return with ido = ', ido
          stop

        case(3)
! Reset options to defaults.  (This is good housekeeping but not
! required for this problem.)
          in = -in
          call iumag ('math', ichap, iput, 50, in, ival)
          inr = -inr
          call sumag ('math', ichap, iput, 20, inr, sval)
          exit Integration_Loop
        case(5)
! Evaluate partials of g(y,y').
          t_y = y; t_ypr = ypr

          t_g = r_diag*t_y + r_off*EOSHIFT(t_y,SHIFT=+1) &
                          + EOSHIFT(r_off*t_y,SHIFT=-1) &
            -  (a_diag*t_ypr + a_off*EOSHIFT(t_ypr,SHIFT=+1) &
                             + EOSHIFT(a_off*t_ypr,SHIFT=-1))
! Move data from assumed size to assumed shape arrays.
          do i=1, n
             wk(ival(1)+i-1) = t_g(i)
          end do
          cycle Integration_Loop

        case(6)
! Evaluate partials of g(y,y').
! Get value of c_j for partials.
          iopt(1) = inr(9)
          call sumag ('math', ichap, iget, 1, iopt, sval)

! Subtract c_j from diagonals to compute (partials for y')*c_j.
! The linear system is tridiagonal.
          t_diag(1:n,1) = r_diag - sval(1)*a_diag
          t_upper(1:n,1) = r_off - sval(1)*a_off
          t_lower = EOSHIFT(t_upper,SHIFT=+1,DIM=1)

          cycle Integration_Loop

        case(7)
! Compute the factorization.
          iopti(1) = s_lin_sol_tri_factor_only
          call lin_sol_tri (t_upper, t_diag, t_lower, &
                  t_sol, iopt=iopti)
          cycle Integration_Loop
        case(8)
! Solve the system.
          iopti(1) = s_lin_sol_tri_solve_only
! Move data from the assumed size to assumed shape arrays.
          t_sol(1:n,1)=wk(ival(1):ival(1)+n-1)

          call lin_sol_tri (t_upper, t_diag, t_lower, &
                    t_sol, iopt=iopti)

! Move data from the assumed shape to assumed size arrays.
          wk(ival(1):ival(1)+n-1)=t_sol(1:n,1)

          cycle Integration_Loop

        case(2)
! Correct initial value to reach u_1 at t=tend.
          u_0 = u_0 - (u_0*y(n/2) - (u_1-u_0)) / (y(n/2) + 1)

! Finish up internally in the integrator.
          ido = 3
          cycle Integration_Loop
      end select
      end do Integration_Loop

   write (*,*) 'The equation u_t = u_xx, with u(0,t) = ', u_0
   write (*,*) 'reaches the value ',u_1, ' at time = ', tend, '.'
   write (*,*) 'Example 4 for LIN_SOL_TRI (operators) is correct.'

   end
```

Operator_ex21

```
      use linear_operators

      implicit none

! This is Example 1 (using operators) for LIN_SVD.

      integer, parameter :: n=32
      real(kind(1d0)), parameter :: one=1d0
      real(kind(1d0)) err
      real(kind(1d0)), dimension(n,n) :: A, U, V, S(n)


! Generate a random n by n matrix.
      A = rand(A)

! Compute the singular value decomposition.
      S=SVD(A, U=U, V=V)

! Check for small residuals of the expression A*V - U*S.
      err = norm((A .x. V) - (U .x. diag(S)))/norm(S)
      if (err  <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for LIN_SVD (operators) is correct.'
      end if

      end
```

Operator_ex22

```
      use linear_operators

      implicit none

! This is Example 2 (using operators) for LIN_SVD.

      integer, parameter :: m=64, n=32, k=4
      real(kind(1d0)), parameter :: one=1.0d0, zero=0.0d0
      real(kind(1d0)) a(m,n), s(n), u(m,m), v(n,n), &
             b(m,k), x(n,k), g(m,k), alpha(k), lamda(k), &
             delta_lamda(k), t_g(n,k), s_sq(n), phi(n,k), &
             phi_dot(n,k), move(k), err

! Generate a random matrix for both A and B.
      A=rand(A); b=rand(b)

! Compute the singular value decomposition.
      S = SVD(A, U=u, V=v)

! Choose alpha so that the lengths of the regularized solutions
! are 0.25 times lengths of the non-regularized solutions.

      g =  u .tx. b; x = v .x. diag(one/S) .x. g(1:n,:)
      alpha = 0.25*sqrt(sum(x**2,DIM=1))
      t_g = diag(S) .x. g(1:n,:); s_sq = s**2; lamda = zero

      solve_for_lamda:  do
         x = one/(spread(s_sq,DIM=2,NCOPIES=k)+ &
                  spread(lamda,DIM=1,NCOPIES=n))

         phi = (t_g*x)**2; phi_dot = -2*phi*x
         delta_lamda = (sum(phi,DIM=1)-alpha**2)/sum(phi_dot,DIM=1)

! Make Newton method correction to solve the secular equations for
! lamda.
         lamda = lamda - delta_lamda

! Test for convergence and quit when it happens.
          if (norm(delta_lamda) <= &
          sqrt(epsilon(one))*norm(lamda)) EXIT solve_for_lamda

! Correct any bad moves to a positive restart.
         move = rand(move); where (lamda < 0) lamda = s(1) * move

      end do solve_for_lamda

! Compute solutions and check lengths.
      x = v .x. (t_g/(spread(s_sq, DIM=2,NCOPIES=k)+ &
                      spread(lamda,DIM=1,NCOPIES=n)))
      err = norm(sum(x**2,DIM=1) - alpha**2)/norm(alpha)**2
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 2 for LIN_SVD (operators) is correct.'
      end if

      end
```

Operator_ex23

```
      use linear_operators

      implicit none

! This is Example 3 (using operators) for LIN_SVD.

      integer, parameter :: n=32
      integer i
      real(kind(1d0)), parameter :: one=1d0
      real(kind(1d0)), dimension(n,n) :: d(2*n,n), x, u_d(2*n,2*n), &
             v_d, v_c, u_c, v_s, u_s, &
             s_d(n), c(n), s(n), sc_c(n), sc_s(n)
      real(kind(1d0)) err1, err2

! Generate random square matrices for both A and B.
! Construct D; A is on the top; B is on the bottom.
      D = rand(D)!   D(1:n,:) = A; D(n+1:,:) = B

! Compute the singular value decompositions used for the GSVD.
      S_D= SVD(D,U=u_d,V=v_d)
      C  = SVD(u_d(1:n, 1:n), u=u_c,v=v_c)
      S  = SVD(u_d(n+1:,1:n), u=u_s,v=v_s)

! Rearrange c(:) so it is non-increasing.  Move singular
! vectors accordingly.  (The use of temporary objects sc_c and
! x is required.)
      sc_c = c(n:1:-1); c = sc_c
      x = u_c(1:n,n:1:-1); u_c = x; x = v_c(1:n,n:1:-1); v_c = x

! The columns of v_c and v_s have the same span.  They are
! equivalent by taking the signs of the largest magnitude values
! positive.
      do i=1, n
         sc_c(i) = sign(one,v_c(sum(maxloc(abs(v_c(1:n,i)))),i))
         sc_s(i) = sign(one,v_s(sum(maxloc(abs(v_s(1:n,i)))),i))
      end do

      v_c = v_c .x. diag(sc_c); u_c =  u_c .x. diag(sc_c)
      v_s = v_s .x. diag(sc_s); u_s =  u_s .x. diag(sc_s)


! In this form of the GSVD, the matrix X can be unstable if D
! is ill-conditioned.
      X = v_d .x. diag(one/s_d) .x. v_c
```

```
! Check residuals for GSVD, A*X = u_c*diag(c_1, ..., c_n), and
! B*X = u_s*diag(s_1, ..., s_n).

      err1 = norm((D(1:n, :) .x. X) - (u_c .x. diag(C)))/s_d(1)
      err2 = norm((D(n+1:,:) .x. X) - (u_s .x. diag(S)))/s_d(1)

      if (err1 <= sqrt(epsilon(one)) .and. &
          err2 <= sqrt(epsilon(one))) then
         write (*,*) 'Example 3 for LIN_SVD (operators) is correct.'
      end if

      end
```

Operator_ex24

```
      use linear_operators

      implicit none

! This is Example 4 (using operators) for LIN_SVD.

      integer i
      integer, parameter :: m=32, n=16, p=10, k=4
      real(kind(1d0)), parameter :: one=1d0
      real(kind(1d0)) log_lamda, log_lamda_t, delta_log_lamda
      real(kind(1d0)) a(m,n), b(m,k), w(m,k), g(m,k), t(n), s(n), &
              s_sq(n), u(m,m), v(n,n), c_lamda(p,k), &
              lamda(k), x(n,k), res(n,k)

! Generate random rectangular matrices for A and right-hand
! sides, b.  Generate random weights for each of the
! right-hand sides.
      A=rand(A); b=rand(b); w=rand(w)

! Compute the singular value decomposition.
      S = SVD(A, U=U, V=V)
      g = U .tx. b; s_sq = s**2

      log_lamda = log(10.*s(1)); log_lamda_t=log_lamda
      delta_log_lamda = (log_lamda - log(0.1*s(n))) / (p-1)

! Choose lamda to minimize the "cross-validation" weighted
! square error.  First evaluate the error at a grid of points,
! uniform in log_scale.

      cross_validation_error:  do i=1, p
         t = s_sq/(s_sq+exp(log_lamda))
         c_lamda(i,:) = sum(w*((b-(U(1:m,1:n) .x. g(1:n,1:k)* &
                        spread(t,DIM=2,NCOPIES=k)))/ &
         (one-(u(1:m,1:n)**2 .x. spread(t,DIM=2,NCOPIES=k))))**2,DIM=1)
         log_lamda = log_lamda - delta_log_lamda
      end do cross_validation_error

! Compute the grid value and lamda corresponding to the minimum.
      do i=1, k
```

```
         lamda(i) = exp(log_lamda_t -  delta_log_lamda* &
                        (sum(minloc(c_lamda(1:p,i)))-1))
      end do

! Compute the solution using the optimum "cross-validation"
! parameters.
      x = V .x. g(1:n,1:k)*spread(s,DIM=2,NCOPIES=k)/ &
                    (spread(s_sq,DIM=2,NCOPIES=k)+ &
                     spread(lamda,DIM=1,NCOPIES=n))
! Check the residuals, using normal equations.
      res = (A .tx. (b - (A .x. x))) - &
            spread(lamda,DIM=1,NCOPIES=n)*x
      if (norm(res)/s_sq(1) <=  sqrt(epsilon(one))) then
         write (*,*) 'Example 4 for LIN_SVD (operators) is correct.'
      end if

      end
```

Operator_ex25

```
      use linear_operators

      implicit none

! This is Example 1 (using operators) for LIN_EIG_SELF.

      integer, parameter :: n=64
      real(kind(1e0)), parameter :: one=1e0
      real(kind(1e0)) :: A(n,n), D(n), S(n)


! Generate a random matrix and from it
! a self-adjoint matrix.
      A = rand(A); A = A + .t.A

! Compute the eigenvalues of the matrix.
      D = EIG(A)

! For comparison, compute the singular values and check for
! any error messages for either decomposition.
      S = SVD(A)

! Check the results:  Magnitude of eigenvalues should equal
! the singular values.

      if (norm(abs(D) - S) <= sqrt(epsilon(one))*S(1)) then
         write (*,*) 'Example 1 for LIN_EIG_SELF (operators) is correct.'
      end if

      end
```

Operator_ex26

```
      use linear_operators

      implicit none

! This is Example 2 (using operators) for LIN_EIG_SELF.

      integer, parameter :: n=8
      real(kind(1e0)), parameter :: one=1e0
      real(kind(1e0)), dimension(n,n) :: A, d(n), v_s

! Generate a random self-adjoint matrix.
      A = rand(A); A = A + .t.A

! Compute the eigenvalues and eigenvectors.
      D = EIG(A,V=v_s)

! Check the results for small residuals.
      if (norm((A .x. v_s) - (v_s .x. diag(D)))/abs(d(1)) <= &
             sqrt(epsilon(one))) then
         write (*,*) 'Example 2 for LIN_EIG_SELF (operators) is correct.'
      end if

      end
```

Operator_ex27

```
      use linear_operators

      implicit none

! This is Example 3 (using operators) for LIN_EIG_SELF.

      integer i
      integer, parameter :: n=64, k=08
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)) err
      real(kind(1d0)), dimension(n,n) :: A, D(n),&
               res(n,k), v(n,k)

! Generate a random self-adjoint matrix.
      A = rand(A); A = A + .t.A

! Compute just the eigenvalues.
      D = EIG(A); V = rand(V)

! Ready options to skip error processing and reset
! tolerance for linear solver.
      allocate (d_invx_options(5))

      do i=1, k

! Use packaged option to reset the value of a small diagonal.
```

```
      d_invx_options(1) = skip_error_processing
      d_invx_options(2) = ix_options_for_lin_sol_gen
        d_invx_options(3) = 2
        d_invx_options(4) = d_options&
        (d_lin_sol_gen_set_small, epsilon(one)*abs(d(i)))
        d_invx_options(5) = d_lin_sol_gen_no_sing_mess

! Compute the eigenvectors with inverse iteration.
         V(1:,i)= (A - EYE(n)*d(i)).ix. V(1:,i)
      end do
      deallocate (d_invx_options)

! Orthogonalize the eigenvectors.
      V = ORTH(V)

! Check the results for both orthogonality of vectors and small
! residuals.

      res(1:k,1:k) = (V .tx. V) - EYE(k)
      err = norm(res(1:k,1:k)); res= (A .x. V) - (V .x. diag(D(1:k)))
      if (err <= sqrt(epsilon(one)) .and. &
         norm(res)/abs(d(1)) <= sqrt(epsilon(one))) then
           write (*,*) 'Example 3 for LIN_EIG_SELF (operators) is correct.'
      end if
      end
```

Operator_ex28

```
      use linear_operators

      implicit none

! This is Example 4 (using operators) for LIN_EIG_SELF.

      integer, parameter :: n=64
      real(kind(1e0)), parameter :: one=1d0
      real(kind(1e0)), dimension(n,n) :: A, B, C, D(n), lambda(n), &
                S(n), vb_d, X, res

! Generate random self-adjoint matrices.
      A = rand(A); A = A + .t.A
      B = rand(B); B = B + .t.B

! Add a scalar matrix so B is positive definite.
      B = B + norm(B)*EYE(n)

! Get the eigenvalues and eigenvectors for B.
      S = EIG(B,V=vb_d)

! For full rank problems, convert to an ordinary self-adjoint
! problem.  (All of these examples are full rank.)
      if (S(n) > epsilon(one)) then
         D = one/sqrt(S)
         C = diag(D) .x. (vb_d .tx. A .x. vb_d) .x. diag(D)
         C = (C + .t.C)/2
```

```
! Get the eigenvalues and eigenvectors for C.
         lambda = EIG(C,v=X)

! Compute and normalize the generalized eigenvectors.
         X = UNIT(vb_d .x. diag(D) .x. X)
         res = (A .x. X) - (B .x. X .x. diag(lambda))

! Check the results.
         if(norm(res)/(norm(A)+norm(B)) <= &
            sqrt(epsilon(one))) then
            write (*,*) 'Example 4 for LIN_EIG_SELF (operators) is correct.'
         end if

      end if

      end
```

Operator_ex29

```
      use linear_operators

      implicit none

! This is Example 1 (using operators) for LIN_EIG_GEN.

      integer, parameter :: n=32
      real(kind(1d0)), parameter :: one=1d0
      real(kind(1d0)) err
      real(kind(1d0)), dimension(n,n) :: A
      complex(kind(1d0)), dimension(n) :: E, E_T, V(n,n)

! Generate a random matrix.
      A = rand(A)

! Compute only the eigenvalues.
      E = EIG(A)

! Compute the decomposition, A*V = V*values,
! obtaining eigenvectors.
      E_T = EIG(A, W = V)

! Use values from the first decomposition, vectors from the
! second decomposition, and check for small residuals.
      err = norm((A .x. V) - (V .x. diag(E)))/&
            (norm(A)+norm(E))

      if (err  <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for LIN_EIG_GEN (operators) is correct.'
      end if

      end
```

Operator_ex30

```
      use linear_operators

      implicit none

! This is Example 2 (using operators) for LIN_EIG_GEN.

      integer i
      integer, parameter :: n=12
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      complex(kind(1d0)), dimension(n) :: a(n,n), b, e, f, fg

      b = rand(b)

! Define the companion matrix with polynomial coefficients
! in the first row.
      A = zero; A = EOSHIFT(EYE(n),SHIFT=1,DIM=2); a(1,1:) = - b

! Compute complex eigenvalues of the companion matrix.
      E = EIG(A)

! Use Horner's method for evaluation of the complex polynomial
! and size gauge at all roots.
      f=one; fg=one
      do i=1, n
         f = f*E + b(i)
         fg = fg*abs(E) + abs(b(i))
      end do

! Check for small errors at all roots.
      if (norm(f/fg) <= sqrt(epsilon(one))) then
         write (*,*) 'Example 2 for LIN_EIG_GEN (operators) is correct.'
      end if

      end
```

Operator_ex31

```
      use linear_operators

      implicit none

! This is Example 3 (using operators) for LIN_EIG_GEN.

      integer, parameter :: n=32, k=2
      real(kind(1e0)), parameter :: one=1e0, zero=0e0
      real(kind(1e0)) a(n,n), b(n,k), x(n,k), h
      complex(kind(1e0)),dimension(n,n) :: W, T, e(n), z(n,k)
      type(s_options) :: iopti(2)

      A = rand(A); b=rand(b)

      iopti(1) = s_lin_eig_gen_out_tri_form
      iopti(2) = s_lin_eig_gen_no_balance

! Compute the Schur decomposition of the matrix.
      call lin_eig_gen(a, e, v=w, &
           tri=t,iopt=iopti)

! Choose a value so that A+h*I is non-singular.
      h = one

! Solve for (A+h*I)x=b using the Schur decomposition.
      z = W .hx. b

! Solve intermediate upper-triangular system with implicit
! additive diagonal, h*I.  This is the only dependence on
! h in the solution process.
      z = (T + h*EYE(n)) .ix. z
! Compute the solution.  It should be the same as x, but will not be
! exact due to rounding errors.  (The quantity real(z,kind(one)) is
! the real-valued answer when the Schur decomposition method is used.)
      z = W .x. z

! Compute the solution by solving for x directly.
      x = (A + EYE(n)*h) .ix. b

! Check that x and z agree approximately.
      if (norm(x-z)/norm(z) <= sqrt(epsilon(one))) then
         write (*,*) 'Example 3 for LIN_EIG_GEN (operators) is correct.'
      end if

      end
```

Operator_ex32

```
      use linear_operators
      implicit none
! This is Example 4 (using operators) for LIN_EIG_GEN.

      integer, parameter :: n=17
      real(kind(1d0)), parameter :: one=1d0
      real(kind(1d0)), dimension(n,n) :: A, C
      real(kind(1d0)) variation(n), eta
      complex(kind(1d0)), dimension(n,n) :: U, V, e(n), d(n)

! Generate a random matrix.
      A = rand(A)

! Compute the eigenvalues, left- and right- eigenvectors.
      D = EIG(A, W=V); E = EIG(.t.A, W=U)

! Compute condition numbers and variations of eigenvalues.
      variation = norm(A)/abs(diagonals( U .hx. V))

! Now perturb the data in the matrix by the relative factors
! eta=sqrt(epsilon) and solve for values again.  Check the
! differences compared to the estimates.  They should not exceed
! the bounds.
      eta = sqrt(epsilon(one))
      C = A + eta*(2*rand(A)-1)*A
      D = EIG(C)

! Looking at the differences of absolute values accounts for
! switching signs on the imaginary parts.
      if (count(abs(d)-abs(e) > eta*variation) == 0) then
         write (*,*) 'Example 4 for LIN_EIG_GEN (operators) is correct.'
      end if
      end
```

Operator_ex33

```
      use linear_operators

      implicit none

! This is Example 1 (using operators) for LIN_GEIG_GEN.

      integer, parameter :: n=32
      real(kind(1d0)), parameter :: one=1d0
      real(kind(1d0)) A(n,n), B(n,n), bta(n), beta_t(n), err
      complex(kind(1d0)) alpha(n), alpha_t(n), V(n,n)

! Generate random matrices for both A and B.
      A = rand(A); B = rand(B)

! Compute the generalized eigenvalues.
      alpha = EIG(A, B=B, D=bta)

! Compute the full decomposition once again, A*V = B*V*values,
! and check for any error messages.
      alpha_t = EIG(A, B=B, D=beta_t, W = V)

! Use values from the first decomposition, vectors from the
! second decomposition, and check for small residuals.
      err = norm((A .x. V .x. diag(bta)) - (B .x. V .x. diag(alpha)),1)/&
            (norm(A,1)*norm(bta,1) + norm(B,1)*norm(alpha,1))
      if (err  <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for LIN_GEIG_GEN (operators) is correct.'
      end if

      end
```

Operator_ex34

```
      use linear_operators

      implicit none

! This is Example 2 (using operators) for LIN_GEIG_GEN.
```

```
      integer, parameter :: n=32
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)) err, alpha(n)
      complex(kind(1d0)), dimension(n,n) :: A, B, C, D, V


! Generate random matrices for both A and B.
      C = rand(C); D = rand(D)
      A = C + .h.C; B = D .hx. D; B = (B + .h.B)/2

      ALPHA = EIG(A, B=B, W=V)

! Check that residuals are small.  Use a real array for  alpha
! since the eigenvalues are known to be real.
      err= norm((A .x. V) - (B .x. V .x. diag(alpha)),1)/&
           (norm(A,1)+norm(B,1)*norm(alpha,1))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 2 for LIN_GEIG_GEN (operators) is correct.'
      end if

      end
```

Operator_ex35

```
      use rand_int
      use eig_int
      use isnan_int
      use norm_int
      use lin_sol_lsq_int

      implicit none

! This is Example 3 (using operators) for LIN_GEIG_GEN.

      integer, parameter :: n=6
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)), dimension(n,n) :: A, B, d_beta(n)
      complex(kind(1d0)) alpha(n)

! Generate random matrices for both A and B.
      A = rand(A); B = rand(B)

! Make columns of A and B zero, so both are singular.
      A(1:n,n) = 0; B(1:n,n) = 0

! Set the option, a larger tolerance than default for lin_sol_lsq.
! Skip showing any error messages.
      allocate(d_eig_options(6))
      d_eig_options(1) = skip_error_processing
      d_eig_options(2) = options_for_lin_geig_gen
      d_eig_options(3) = 3
        d_eig_options(4) = d_lin_geig_gen_for_lin_sol_lsq
        d_eig_options(5) = 1
        d_eig_options(6) = d_options(d_lin_sol_lsq_set_small,&
                           sqrt(epsilon(one))*norm(B,1))

! Compute the generalized eigenvalues.
      ALPHA = EIG(A, B=B, D=d_beta)

! See if singular DAE system is detected.
      if (isNaN(ALPHA)) then
         write (*,*) 'Example 3 for LIN_GEIG_GEN (operators) is correct.'
      end if
! Clean up allocated option arrays for good housekeeping.
      deallocate(d_eig_options)
      end
```

Operator_ex36

```
      use linear_operators

      implicit none

! This is Example 4 for LIN_GEIG_GEN (using operators).

      integer, parameter :: n=32
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)) a(n,n), b(n,n), bta(n), err
      complex(kind(1d0)) alpha(n), v(n,n)

! Generate random matrices for both A and B.
      A = rand(A); B = rand(B)


! Set the option, a larger tolerance than default for lin_sol_lsq.
      allocate(d_eig_options(6))
      d_eig_options(1) = options_for_lin_geig_gen
      d_eig_options(2) = 4
        d_eig_options(3) = d_lin_geig_gen_for_lin_sol_lsq
        d_eig_options(4) = 2
        d_eig_options(5) = d_options(d_lin_sol_lsq_set_small,&
                           sqrt(epsilon(one))*norm(B,1))
        d_eig_options(6) = d_lin_sol_lsq_no_sing_mess

! Compute the generalized eigenvalues.
      alpha = EIG(A, B=B, D=bta, W=V)

! Check the residuals.
      err = norm((A .x. V .x. diag(bta)) - (B .x. V .x. diag(alpha)),1)/&
            (norm(A,1)*norm(bta,1)+norm(B,1)*norm(alpha,1))

      if (err  <= sqrt(epsilon(one))) then
         write (*,*) 'Example 4 for LIN_GEIG_GEN (operators) is correct.'
      end if
! Clean up the allocated array.  This is good housekeeping.
      deallocate(d_eig_options)
      end
```

Operator_ex37

```
      use rand_gen_int
      use fft_int
      use ifft_int
      use linear_operators

      implicit none

! This is Example 4 for FAST_DFT (using operators).

      integer j
      integer, parameter :: n=40
      real(kind(1e0)) :: err, one=1e0
      real(kind(1e0)), dimension(n) :: a, b, c, yy(n,n)
      complex(kind(1e0)), dimension(n) ::  f

! Generate two random periodic sequences 'a' and 'b'.
      a=rand(a); b=rand(b)

! Compute the convolution 'c' of 'a' and 'b'.
      yy(1:,1)=b
      do j=2,n
        yy(2:,j)=yy(1:n-1,j-1)
        yy(1,j)=yy(n,j-1)
      end do

      c=yy .x. a

! Compute f=inverse(transform(a)*transform(b)).
      f=ifft(fft(a)*fft(b))

! Check the Convolution Theorem:
! inverse(transform(a)*transform(b)) = convolution(a,b).
      err = norm(c-f)/norm(c)
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 4 for FAST_DFT (operators) is correct.'
      end if

      end
```

Parallel Examples

MPI REQUIRED

This section presents a variation of key examples listed above or in other parts of the document. In all cases the examples appear to be simple, use parallel computing, deliver results to the root, and have been tested for correctness by validating small residuals or other first principles. Program names are `parallel_exnn`, where `nn=01,02,...` The numerical digit part of the name matches the example number.

Parallel Examples 1-2 comments

These show the box data type used for solving several systems and then checking the results using matrix products and norms or other mathematical relationships. Note the first call to the function `MP_SETUP()` that initiates MPI. The call to the function `MP_SETUP('Final')` shuts down MPI and retrieves any error messages from the nodes. It is only here that error messages will print, in reverse node order, at the root node. Note that the results are checked for correctness at the root node. (This is common to all the parallel examples.)

Parallel Example 1

```
    use linear_operators
    use mpi_setup_int

    implicit none

This is Parallel Example 1 for .ix., with box data types
and functions.

    integer, parameter :: n=32, nr=4
    real(kind(1e0)) :: one=1e0
    real(kind(1e0)), dimension(n,n,nr) :: A, b, x, err(nr)

Setup for MPI.
    MP_NPROCS=MP_SETUP()

Generate random matrices for A and b:
    A = rand(A); b=rand(b)

Compute the box solution matrix of Ax = b.
    x = A .ix. b

Check the results.
    err = norm(b - (A .x. x))/(norm(A)*norm(x)+norm(b))
    if (ALL(err <= sqrt(epsilon(one))) .and. MP_RANK == 0) &
      write (*,*) 'Parallel Example 1 is correct.'

See to any error messages and quit MPI.
    MP_NPROCS=MP_SETUP('Final')

    end
```

Parallel Example 2

```
      use linear_operators
      use mpi_setup_int

      implicit none

! This is Parallel Example 2 for .i. and det() with box
! data types, operators and functions.

      integer, parameter :: n=32, nr=4
      integer J
      real(kind(1e0)) :: one=1e0
      real(kind(1e0)), dimension(nr) :: err, det_A, det_i
      real(kind(1e0)), dimension(n,n,nr) :: A, inv, R, S

! Setup for MPI.
      MP_NPROCS=MP_SETUP()
! Generate a random matrix.
      A = rand(A)
! Compute the matrix inverse and its determinant.
      inv = .i.A; det_A = det(A)
! Compute the determinant for the inverse matrix.
      det_i = det(inv)
! Check the quality of both left and right inverses.
      DO J=1,nr; R(:,:,J)=EYE(N); END DO

      S=R; R=R-(A .x. inv); S=S-(inv .x. A)
      err = (norm(R)+norm(S))/cond(A)
      if (ALL(err <= sqrt(epsilon(one)) .and. &
        abs(det_A*det_i - one) <= sqrt(epsilon(one)))&
       .and. MP_RANK == 0) &
        write (*,*) 'Parallel Example 2 is correct.'

! See to any error messages and quit MPI.
      MP_NPROCS=MP_SETUP('Final')

      end
```

Parallel Example 3

This example shows the box data type used while obtaining an accurate solution of several systems. Important in this example is the fact that only the root will achieve convergence, which controls program flow out of the loop. Therefore the nodes must share the root’s view of convergence, and that is the reason for the broadcast of the update from root to the nodes. Note that when writing an explicit call to an MPI routine there must be the line `INCLUDE ‘mpif.h’`, placed just after the `IMPLICIT NONE` statement. Any number of nodes can be used.

```
      use linear_operators
      use mpi_setup_int
```

```
    implicit none
    INCLUDE 'mpif.h'

This is Parallel Example 3 for .i. and iterative
refinement with box date types, operators and functions.
    integer, parameter :: n=32, nr=4
    integer IERROR
    real(kind(1e0)) :: one=1e0, zero=0e0
    real(kind(1e0)) :: A(n,n,nr), b(n,1,nr), x(n,1,nr)
    real(kind(1e0)) change_old(nr), change_new(nr)
    real(kind(1d0)) :: d_zero=0d0, c(n,1,nr), D(n,n,nr), y(n,1,nr)

Setup for MPI.
    MP_NPROCS=MP_SETUP()

Generate a random matrix and right-hand side.
    A = rand(A); b= rand(b)

Save double precision copies of the matrix and right-hand side.
    D = A
    c = b

Get single precision inverse to compute the iterative refinement.
    A = .i. A

Start solution at zero.  Update it to a more accurate solution
with each iteration.
    y = d_zero
    change_old = huge(one)

    ITERATIVE_REFINEMENT: DO

Compute the residual with higher accuracy than the data.
       b = c - (D .x. y)

Compute the update in single precision.
       x = A .x. b
       y = x + y
       change_new = norm(x)

All processors must share the root's test of convergence.
       CALL MPI_BCAST(change_new, nr, MPI_REAL, 0, &
         MP_LIBRARY_WORLD, IERROR)

Exit when changes are no longer decreasing.
       if (ALL(change_new >= change_old)) exit iterative_refinement
       change_old = change_new
    end DO ITERATIVE_REFINEMENT

      IF(MP_RANK == 0) write (*,*) 'Parallel Example 3 is correct.'

See to any error messages and quit MPI.
    MP_NPROCS=MP_SETUP('Final')
    end
```

Parallel Example 4

Here an alternate node is used to compute the majority of a single application, and the user does not need to make any explicit calls to MPI routines. The time-consuming parts are the evaluation of the eigenvalue-eigenvector expansion, the solving step, and the residuals. To do this, the rank-2 arrays are changed to a box data type with a unit third dimension. This uses parallel computing. The node priority order is established by the initial function call, `MP_SETUP(n)`. The root is restricted from working on the box data type by assigning `MPI_ROOT_WORKS=.false.` This example anticipates that the most efficient node, other than the root, will perform the heavy computing. Two nodes are required to execute.

```
    use linear_operators
    use mpi_setup_int

    implicit none

This is Parallel Example 4 for matrix exponential.
The box dimension has a single rack.
    integer, parameter :: n=32, k=128, nr=1
    integer i
    real(kind(1e0)), parameter :: one=1e0, t_max=one, delta_t=t_max/(k-1)
    real(kind(1e0)) err(nr), sizes(nr), A(n,n,nr)
    real(kind(1e0)) t(k), y(n,k,nr), y_prime(n,k,nr)
    complex(kind(1e0)), dimension(n,nr) :: x(n,n,nr), z_0, &
      Z_1(n,nr,nr), y_0, d

Setup for MPI.  Establish a node priority order.
Restrict the root from significant computing.
Illustrates using the 'best' performing node that
is not the root for a single task.
    MP_NPROCS=MP_SETUP(n)

    MPI_ROOT_WORKS=.false.

Generate a random coefficient matrix.
    A = rand(A)

Compute the eigenvalue-eigenvector decomposition
of the system coefficient matrix on an alternate node.
    D = EIG(A, W=X)

Generate a random initial value for the ODE system.
    y_0 = rand(y_0)

Solve complex data system that transforms the initial
values, X z_0=y_0.

    z_1= X .ix. y_0 ; z_0(:,nr) = z_1(:,nr,nr)

The grid of points where a solution is computed:
    t = (/(i*delta_t,i=0,k-1)/)
```

```
 Compute y and y' at the values t(1:k).
 With the eigenvalue-eigenvector decomposition AX = XD, this
 is an evaluation of EXP(A t)y_0 = y(t).
     y = X .x.exp(spread(d(:,nr),2,k)*spread(t,1,n))*spread(z_0(:,nr),2,k)

 This is y', derived by differentiating y(t).
     y_prime  = X .x. &
pread(d(:,nr),2,k)*exp(spread(d(:,nr),2,k)*spread(t,1,n))* &
               spread(z_0(:,nr),2,k)

 Check results. Is  y' - Ay = 0?
     err = norm(y_prime-(A .x. y))
     sizes=norm(y_prime)+norm(A)*norm(y)
     if (ALL(err <= sqrt(epsilon(one))*sizes) .and. MP_RANK == 0) &
       write (*,*) 'Parallel Example 4 is correct.'

 See to any error messages and quit MPI.
     MP_NPROCS=MP_SETUP('Final')

     end
```

Parallel Example 5-6 comments

The computations performed in these examples are for linear least-squares solutions. There is use of the box data type and MPI. Otherwise these are similar to Parallel Examples 1-2 except they use alternate operators and functions. Any number of nodes can be used.

Parallel Example 5

```
     use linear_operators
     use mpi_setup_int

     implicit none

 This is Parallel Example 5 using box data types, operators
 and functions.

     integer, parameter :: m=64, n=32, nr=4
     real(kind(1e0)) :: one=1e0, err(nr)
     real(kind(1e0)), dimension(n,n,nr) :: A, b, x
     real(kind(1e0)), dimension(m,n,nr) :: C, d

 Setup for MPI.
     mp_nprocs = mp_setup()

 Generate two rectangular random matrices, only
```

```
at the root node.
    if (mp_rank == 0) then
      C = rand(C); d=rand(d)
    endif

Form the normal equations for the rectangular system.
    A = C .tx. C; b = C .tx. d

Compute the solution for Ax = b.
    x = A .ix. b

Check the results.
    err = norm(b - (A .x. x))/(norm(A)+norm(b))
    if (ALL(err <= sqrt(epsilon(one))) .AND. MP_RANK == 0) &
       write (*,*) 'Parallel Example 5 is correct.'

See to any error messages and quit MPI.
    mp_nprocs = mp_setup('Final')

    end
```

Parallel Example 6

```
    use linear_operators
    use mpi_setup_int

    implicit none

This is Parallel Example 6 for box data types, operators and
functions.

    integer, parameter :: m=64, n=32, nr=4
    real(kind(1e0)) :: one=1e0, zero=0e0, err(nr)
    real(kind(1e0)), dimension(m,n,nr) :: C, d(m,1,nr)
    real(kind(1e0)), dimension(n,n,nr) :: A, cov
    real(kind(1e0)), dimension(n,1,nr) :: b, x

Setup for MPI:
    mp_nprocs=mp_setup()

Generate a random rectangular matrix and right-hand side.
    if(mp_rank == 0) then
       C = rand(C); d=rand(d)
    endif

Form the normal equations for the rectangular system.
    A = C .tx. C; b = C .tx. d
    COV = .i. CHOL(A); COV = COV .xt. COV

Compute the least-squares solution.
     x = C .ix. d

Compare with solution obtained using the inverse matrix.
    err = norm(x - (COV .x. b))/norm(cov)
```

```
Check the results.
    if (ALL(err <= sqrt(epsilon(one))) .and. mp_rank == 0) &
       write (*,*) 'Parallel Example 6 is correct.'

See to any eror messages and quit MPI
    mp_nprocs=mp_setup('Final')

    end
```

Parallel Example 7

In this example alternate nodes are used for computing with the `EIG()` function. Inverse iteration is used to obtain eigenvectors for the second most dominant eigenvalue for each rack of the box. The factorization and solving steps for the eigenvectors are executed only at the root node.

```
    use linear_operators
    use mpi_setup_int

    implicit none

This is Parallel Example 7 for box data types, operators
and functions.

    integer tries, nrack
    integer, parameter :: m=8, n=4, k=2, nr=4
    integer ipivots(n+1)
    real(kind(1d0)) :: one=1D0, err(nr), E(n,nr)
    real(kind(1d0)), dimension(m,n,nr) ::  C
    real(kind(1d0)), dimension(n,n,nr) ::  A, ATEMP
    real(kind(1d0)), dimension(n,1,nr) ::  b, x
    type(d_options) :: iopti(4)
    logical, dimension(nr) :: results_are_true

Setup for MPI:
    mp_nprocs = mp_setup()

Generate a random rectangular matrix.
    if (mp_rank == 0) C = rand(C)

Generate a random right hand side for use in the
inverse iteration.
    if (mp_rank == 0) b = rand(b)

Compute a positive definite matrix.
    A = C .tx. C; A = (A + .t.A)/2

Obtain just the eigenvalues.
    E = EIG(A)

    ATEMP = A
```

```
Compute A-eigenvalue*I as the coefficient matrix.
Use eigenvalue number k.

    do nrack = 1,nr
       IF(MP_RANK > 0) EXIT
Use packaged option to reset the value of a small diagonal.

       iopti(1) = d_options(d_lin_sol_self_set_small,&
               epsilon(one)*abs(E(1,nrack)))

Use packaged option to save the factorization.
       iopti(2) = d_lin_sol_self_save_factors

Suppress error messages and stopping due to singularity
of the matrix, which is expected.
       iopti(3) = d_lin_sol_self_no_sing_mess
       iopti(4) = 0
       A(:,:,nrack) = A(:,:,nrack) - E(k,nrack)*EYE(n)

       do tries=1,2
          call lin_sol_self(A(:,:,nrack), &
                   b(:,:,nrack), x(:,:,nrack), &
                   pivots=ipivots, iopt=iopti)
When code is re-entered, the already computed factorization
is used.
          iopti(4) = d_lin_sol_self_solve_A

Reset right-hand side in the direction of the eigenvector.
          B(:,:,nrack) = UNIT(x(:,:,nrack))
       end do

       end do

Normalize the eigenvector.

    IF(MP_RANK == 0) x = UNIT(x)


Check the results.
    b = ATEMP .x. x

    do nrack = 1,nr
       err(nrack) =  &
         dot_product(x(1:n,1,nrack), b(1:n,1,nrack)) - E(k,nrack)
       results_are_true(nrack) = &
         (abs(err(nrack)) <= sqrt(epsilon(one))*E(1,nrack))
    enddo

Check the results.
    if (ALL(results_are_true) .and. MP_RANK == 0) &
      write (*,*) 'Parallel Example 7 is correct.'
```

```
See to any error messages and quit MPI.
    mp_nprocs = mp_setup('Final')
    end
```

Parallel Example 8

This example, similar to Parallel Example 3, shows the box data type used while obtaining an accurate solution of several linear least-squares systems. Computation of the residuals for the box data type is executed in parallel. Only the root node performs the factorization and update step during iterative refinement.

```
    use linear_operators
    use mpi_setup_int

    implicit none

    INCLUDE 'mpif.h'

This is Parallel Example 8.  All nodes share in
just part of the work.

    integer, parameter :: m=8, n=4 , nr=4
    real(kind(1e0)) :: one=1e0, zero=0e0
    real(kind(1d0)) :: d_zero=0d0
    integer ipivots((n+m)+1), ierror, nrack
    real(kind(1e0)) A(m,n,nr), b(m,1,nr), F(n+m,n+m,nr),&
          g(n+m,1,nr), h(n+m,1,nr)
    real(kind(1e0)) change_new(nr), change_old(nr)
    real(kind(1d0)) c(m,1,nr), D(m,n,nr), y(n+m,1,nr)
    type(s_options) ::  iopti(2)

Setup for MPI:
    mp_nprocs=mp_setup()

Generate a random matrix and right-hand side.
    if(mp_rank == 0) then
       A = rand(A); b = rand(b)
    endif

Save double precision copies of the matrix and right hand side.
    D = A; c = b

Fill in augmented matrix for accurately solving the least-squares
problem using iterative refinement.
    F = zero
    do nrack = 1,nr
       F(1:m,1:m,nrack)=EYE(m)
    enddo
    F(1:m,m+1:,:) = A; F(m+1:,1:m,:) = .t. A
```

```
 Start solution at zero.
     y = d_zero
     change_old = huge(one)

 Use packaged option to save the factorization.
     iopti(1) = s_lin_sol_self_save_factors
     iopti(2) = 0
     h = zero

        ITERATIVE_REFINEMENT: DO
           g(1:m,:,:) = c(1:m,:,:) - y(1:m,:,:) &
                           - (D .x.  y(m+1:m+n,:,:))
           g(m+1:m+n,:,:) = - D .tx. y(1:m,:,:)
           if(mp_rank == 0) then
              do nrack = 1,nr
                 call lin_sol_self(F(:,:,nrack), &
              g(:,:,nrack), h(:,:,nrack), pivots=ipivots, iopt=iopti)
              enddo
              y = h + y
           endif

           change_new = norm(h)

 All processors share the root's test for convergence
           call mpi_bcast(change_new, nr, MPI_REAL,0, MP_LIBRARY_WORLD,
ERROR)

 Exit when changes are no longer decreasing.
           if (ALL(change_new >= change_old) )&
                   exit iterative_refinement
           change_old = change_new

 Use option to re-enter code with factorization saved; solve only.
           iopti(2) = s_lin_sol_self_solve_A
        end do iterative_refinement

     if(mp_rank == 0)&
       write (*,*) 'Parallel Example 8 is correct.'

 See to any error message and quit MPI.
     mp_nprocs=mp_setup('Final')

     end
```

Parallel Example 9

This is a variation of Parallel Example 8. A single problem is converted to a box data type with one rack. The use of the function call `MP_SETUP(M+N)` allocates and defines the array `MPI_NODE_PRIORITY(:)`, the node priority order. By setting `MPI_ROOT_WORKS=.false.`, the computation of the residual is off-loaded to the node with highest priority, wherein we expect the results to be computed the fastest. The remainder of the computation, including the factorization and solve step, are executed at the root node. This example requires two nodes to execute.

```
se linear_operators
     use mpi_setup_int
     implicit none

     INCLUDE 'mpif.h'

 This is Parallel Example 9, showing iterative
 refinement with only one non-root node working.
 There is only one problem in this example.
     integer, parameter :: m=8, n=4, nr=1
     real(kind(1e0)) :: one=1e0, zero=0e0
     real(kind(1d0)) :: d_zero=0d0
     integer ipivots((n+m)+1), nrack, ierror
     real(kind(1e0)) A(m,n,nr), b(m,1,nr), F(n+m,n+m,nr),&
           g(n+m,1,nr), h(n+m,1,nr)
     real(kind(1e0)) change_new(nr), change_old(nr)
     real(kind(1d0)) c(m,1,nr), D(m,n,nr), y(n+m,1,nr)
     type(s_options) ::  iopti(2)

 Setup for MPI.  Establish a node priority order.
 Restrict the root from significant computing.
 Illustrates the "best" performing non-root node
 computing a single task.
     mp_nprocs=mp_setup(m+n)

     MPI_ROOT_WORKS = .false.

 Generate a random matrix and right-hand side.
     A = rand(A); b = rand(b)

 Save double precision copies of the matrix and right hand side.
     D = A; c = b

 Fill in augmented matrix for accurately solving the least-squares
 problem using iterative refinement.
     F = zero;

     do nrack = 1,nr; F(1:m,1:m,nrack)=EYE(m); end do

     F(1:m,m+1:,:) = A; F(m+1:,1:m,:) = .t. A

 Start solution at zero.
     y = d_zero
     change_old = huge(one)

 Use packaged option to save the factorization.
     iopti(1) = s_lin_sol_self_save_factors
     iopti(2) = 0

     h = zero
     ITERATIVE_REFINEMENT: DO
```

```
        g(1:m,:,:) = c(1:m,:,:) - y(1:m,:,:) - (D .x. y(m+1:m+n,:,:))
        g(m+1:m+n,:,:) = - D .tx. y(1:m,:,:)
        IF (MP_RANK == 0) THEN

          call lin_sol_self(F(:,:,nr), g(:,:,nr), &
            h(:,:,nr), pivots=ipivots, iopt=iopti)

        y = h + y
        END IF

        change_new = norm(h)

 All processors share the root's test for convergence
        call mpi_bcast(change_new, nr, mpi_real, 0, mp_library_world,
error)

 Exit when changes are no longer decreasing.
        if (ALL(change_new >= change_old))&
                   exit ITERATIVE_REFINEMENT
        change_old = change_new

 Use option to re-enter code with factorization saved; solve only.
        iopti(2) = s_lin_sol_self_solve_A
     end do ITERATIVE_REFINEMENT

     if(mp_rank == 0) &
     write (*,*) 'Parallel Example 9 is correct.'
 See to any error messages and quit MPI.
     mp_nprocs = mp_setup('Final')
     end
```

Parallel Example 10

This illustrates the computation of a box data type least-squares polynomial data fitting problem.   The problem is generated at the root node. The alternate nodes are used to solve the least-squares problems. Results are checked at the root node. Any number of nodes can be used.

```
     use linear_operators
     use mpi_setup_int
     use Numerical_Libraries, only : DCONST
     implicit none

 This is Parallel Example 10 for .ix..
     integer i, nrack
     integer, parameter :: m=128, n=8, nr=4
     real(kind(1d0)), parameter :: one=1d0, zero=0d0
     real(kind(1d0)) A(m,0:n,nr), c(0:n,1,nr), pi_over_2, &
       x(m,1,nr), y(m,1,nr), u(m,1,nr), v(m,1,nr), &
       w(m,1,nr), delta_x

 Setup for MPI:
     mp_nprocs = mp_setup()

 Generate a random grid of points and transform
```

```
to the interval (-1,1).
    if(mp_rank == 0) x = rand(x)
    x = x*2 - one

Get the constant 'PI'/2 from IMSL Numerical Libraries.
    pi_over_2 = DCONST((/'PI'/))/2

Generate function data on the grid.
    y = exp(x) + cos(pi_over_2*x)

Fill in the least-squares matrix for the Chebyshev polynomials.
    A(:,0,:) = one; A(:,1,:) = x(:,1,:)

    do i=2, n
       A(:,i,:) = 2*x(:,1,:)*A(:,i-1,:) - A(:,i-2,:)
    end do

Solve for the series coefficients.
    c = A .ix. y

Generate an equally spaced grid on the interval.
    delta_x = 2/real(m-1,kind(one))
    do nrack = 1,nr
       x(:,1,nrack) = (/(-one + i*delta_x,i=0,m-1)/)
    enddo

Evaluate residuals using backward recurrence formulas.
    u = zero; v = zero
    do nrack =1,nr
       do i=n, 0, -1
          w(:,:,nrack) = 2*x(:,:,nrack)*u(:,:,nrack) - &
            v(:,:,nrack) + c(i,1,nrack)
          v(:,:,nrack) = u(:,:,nrack)
          u(:,:,nrack) = w(:,:,nrack)
       end do
    enddo

Compute residuals at the grid:
    y = exp(x) + cos(pi_over_2*x) - (u-x*v)

Check that n+1 sign changes in the residual curve occur.
    x = one
    x = sign(x,y)

    if (count(x(1:m-1,1,:) /= x(2:m,1,:)) >= n+1) then
       if(mp_rank == 0)&
       write (*,*) 'Parallel Example 10 is correct.'
    end if

See to any error messages and exit MPI.
    MP_NPROCS = MP_SETUP('Final')
    end
```

Parallel Example 11

In this example a single problem is elevated by using the box data type with one rack. The function call MP_SETUP(M) may take longer to compute than the computation of the generalized inverse, which follows. Other methods for determining the node priority order, perhaps based on specific knowledge of the network environment, may be better suited for this application. This example requires two nodes to execute.

```
    use linear_operators
    use mpi_setup_int
    use Numerical_Libraries, only : DCONST
    implicit none

This is Parallel Example 11 using a priority order with
only the fastest alternate node working.

    integer i
    integer, parameter :: m=128, n=8, nr=1
    real(kind(1d0)), parameter :: one=1d0, zero=0d0
    real(kind(1d0)) A(m,0:n,nr), c(0:n,1,nr), pi_over_2, x(m), &
      y(m,1,nr), u(m), v(m), w(m), delta_x, inv(0:n, m, nr)

Setup for MPI.  Create a priority order list.  Force the
problem to work on the fastest non-root machine.
    mp_nprocs = mp_setup(m)
    MPI_ROOT_WORKS = .false.

Generate an array of equally spaced points on the interval (-1,1).
    delta_x = 2/real(m-1,kind(one))
    x = (/(-one + i*delta_x,i=0,m-1)/)

Get the constant 'PI'/2 from IMSL Numerical Libraries.
    pi_over_2 = DCONST((/'PI'/))/2

Compute data values on the grid.
    y(:,1,1) = exp(x) + cos(pi_over_2*x)

Fill in the least-squares matrix for the Chebyshev polynomials.
    A(:,0,1) = one
    A(:,1,1) = x

    do i=2, n
       A(:,i,1) = 2*x*A(:,i-1,1) - A(:,i-2,1)
    end do

Compute the generalized inverse of the least-squares matrix.
Compute the series coefficients using the generalized inverse
as 'smoothing formulas.'
    inv = .i. A; c = inv .x. y
Evaluate residuals using backward recurrence formulas.

    u = zero
    v = zero
    do i=n, 0, -1
       w = 2*x*u - v + c(i,1,1)
       v = u
       u = w
    end do
```

```
Compute residuals at the grid:
    y(:,1,1) = exp(x) + cos(pi_over_2*x) - (u-x*v)

Check that n+2 sign changes in the residual curve occur.
    x = one; x = sign(x,y(:,1,1))

    if (count(x(1:m-1) /= x(2:m)) == n+2) then
       if(mp_rank == 0)&
       write (*,*) 'Parallel Example 11 is correct.'
    end if

See to any error messages and exit MPI
    mp_nprocs = mp_setup('Final')
    end
```

Parallel Example 12

his illustrates a surface fitting problem using radial basis functions and a box data ype. It is of interest because this problem fits three component functions of the ame form in a space of dimension two. The racks of the box represent the eparate problems for the three coordinate functions. The coefficients are obtained vith the **.ix.** operator. When the least-squares fitting process requires more laborate software, it may be necessary to send the data to the nodes, compute, and end the results back to the root. See Parallel Example 18 for more details. Any umber of nodes can be used.

```
    use linear_operators
    use mpi_setup_int
    implicit none

This is Parallel Example 12 for
.ix. , NORM, .tx. and .x. operators.
    integer i, j, nrack
    integer, parameter :: m=128, n=32, k=2, n_eval=16, nr=3
    real(kind(1d0)), parameter :: one=1d0, delta_sqr=1d0
    real(kind(1d0)) A(m,n,nr), b(m,1,nr), c(n,1,nr), p(k,m,nr), q(k,n,nr)

Setup for MPI:
    mp_nprocs = mp_setup()

Generate a random set of data and center points in k=2 space.
    if( mp_rank == 0) then
       p = rand(p); q=rand(q)

Compute the coefficient matrix for the least-squares system.
       do nrack=1,nr
          A(:,:,nrack) = sqrt(sum((spread(p(:,:,nrack),3,n) - &
            spread(q(:,:,nrack),2,m))**2,dim=1) + delta_sqr)

Compute the right-hand side of function values.
          b(:,1,nrack) = exp(-sum(p(:,:,nrack)**2,dim=1))
       enddo
    endif

Compute the least-squares solution.  An error message due
to rank deficiency is ignored with the flags:

    allocate (d_invx_options(1))
    d_invx_options(1)=skip_error_processing
    c = A .ix. b

Check the results.
    if (ALL(norm(A .tx. (b - (A .x. c)))/(norm(A)+norm(c)) &
        <= sqrt(epsilon(one)))) then
       if(mp_rank == 0) &
          write (*,*) 'Parallel Example 12 is correct.'
    end if

Unload option type for good housekeeping.
    deallocate (d_invx_options)

See to any error messages and quit MPI.

    mp_nprocs = mp_setup('Final')

    end
```

Parallel Example 13

Here least-squares problems are solved, each with an equality constraint that the variables sum to the value one. A box data type is used and the solution obtained with the **.ix.** operator. Any number of nodes can be used.

```
   use linear_operators
    use mpi_setup_int
    implicit none

This is Parallel Example 13 for .ix. and NORM

    integer, parameter :: m=64, n=32, nr=4
    real(kind(1e0)) :: one=1e0, A(m+1,n,nr), b(m+1,1,nr), x(n,1,nr)


Setup for MPI:
    mp_nprocs=mp_setup()

    if(mp_rank == 0) then
Generate a random matrix and right-hand side.
       A=rand(A); b = rand(b)

Heavily weight desired constraint.  All variables sum to one.
       A(m+1,:,:) =   one/sqrt(epsilon(one))
       b(m+1,:,:) =   one/sqrt(epsilon(one))

    endif
```

```
Compute the least-squares solution with this heavy weight.
    x = A .ix. b

Check the constraint.
    if (ALL(abs(sum(x(:,1,:),dim=1) - one)/norm(x) &
          <= sqrt(epsilon(one)))) then
       if(mp_rank == 0) &
       write (*,*) 'Parallel Example 13 is correct.'
    endif

See to any error messages and exit MPI
    mp_nprocs=mp_setup('Final')

    end
```

Parallel Example 14

Systems of least-squares problems are solved, but now using the `SVD()` function. A box data type is used. This is an example which uses optional arguments and a generic function overloaded for parallel execution of a box data type. Any number of nodes can be used.

```
    use linear_operators
    use mpi_setup_int
    implicit none

This is Parallel Example 14
for SVD, .tx. , .x. and NORM.
    integer, parameter :: m=128, n=32, nr=4
    real(kind(1d0)) :: one=1d0, err(nr)
    real(kind(1d0)) A(m,n,nr), b(m,1,nr), x(n,1,nr), U(m,m,nr), &
      V(n,n,nr), S(n,nr), g(m,1,nr)

Setup for MPI:
    mp_nprocs=mp_setup()

    if(mp_rank == 0) then
Generate a random matrix and right-hand side.
       A = rand(A); b = rand(b)
    endif

Compute the least-squares solution matrix of Ax=b.
    S = SVD(A, U = U, V = V)
    g = U .tx. b
    x = V .x. (diag(one/S) .x. g(1:n,:,:))

Check the results.
    err = norm(A .tx. (b - (A .x. x)))/(norm(A)+norm(x))
    if (ALL(err <= sqrt(epsilon(one)))) then
       if(mp_rank == 0) &
       write (*,*) 'Parallel Example 14 is correct.'
    end if

See to any error messages and quit MPI
```

```
    mp_nprocs = mp_setup('Final')

    end
```

Parallel Example 15

A “Polar Decomposition” of several matrices are computed. The box data type and the `SVD()` function are used. Orthogonality and small residuals are checked to verify that the results are correct.

```
    use linear_operators
    use mpi_setup_int
    implicit none

This is Parallel Example 15 using operators and
functions for a polar decomposition.
    integer, parameter :: n=33, nr=3
    real(kind(1d0)) :: one=1d0, zero=0d0
    real(kind(1d0)),dimension(n,n,nr) :: A, P, Q, &
           S_D(n,nr), U_D, V_D
    real(kind(1d0)) TEMP1(nr), TEMP2(nr)

Setup for MPI:
    mp_nprocs = mp_setup()

Generate a random matrix.
    if(mp_rank == 0) A = rand(A)

Compute the singular value decomposition.
    S_D = SVD(A, U=U_D, V=V_D)

Compute the (left) orthogonal factor.
    P = U_D .xt. V_D

Compute the (right) self-adjoint factor.
    Q = V_D .x. diag(S_D) .xt. V_D
Check the results for orthogonality and
small residuals.
    TEMP1 = NORM(spread(EYE(n),3,nr) - (p .xt. p))
    TEMP2 = NORM(A -(P .X. Q)) / NORM(A)
    if (ALL(TEMP1 <= sqrt(epsilon(one))) .and. &
        ALL(TEMP2 <= sqrt(epsilon(one)))) then
          if(mp_rank == 0)&
          write (*,*) 'Parallel Example 15 is correct.'
    end if

See to any error messages and exit MPI.
    mp_nprocs = mp_setup('Final')

    end
```

Parallel Example 16

A compute-intensive single task, in this case the singular values decomposition of a matrix, is computed and partially reconstructed with matrix products. This result is sent back to the root node. The node of highest priority, not the root, is used for the computation except when only the root is available.

```
     use linear_operators
     use mpi_setup_int
     implicit none
     INCLUDE 'mpif.h'

 This is Parallel Example 16 for SVD.
     integer i, j, IERROR, BEST
     integer, parameter :: n=32
     real(kind(1e0)), parameter :: half=5e-1, one=1e0, zero=0e0
     real(kind(1e0)), dimension(n,n) :: A, S(n), U, V, C
     integer k, STATUS(MPI_STATUS_SIZE)

 Setup for MPI:
     mp_nprocs = mp_setup(n)

EST=1
LOCK: DO

 Fill in value one for points inside the circle,
 zero on the outside.
     A = zero
     DO i=1, n
        DO j=1, n
           if ((i-n/2)**2 + (j-n/2)**2 <= (n/4)**2) A(i,j) = one
        END DO
     END DO
F(MP_NPROCS > 1 .and. MPI_NODE_PRIORITY(1) == 0) BEST=2

 Only the most effective node does this job.
 The rest set idle.
  IF(MP_RANK /= MPI_NODE_PRIORITY(BEST)) EXIT BLOCK

 Compute the singular value decomposition.
     S = SVD(A, U=U, V=V)

 How many terms, to the nearest integer, match the circle?
     k = count(S > half)
     C = U(:,1:k) .x. diag(S(1:k)) .xt. V(:,1:k)

 If root is not the most efficient node, send C back.
     IF(MPI_NODE_PRIORITY(BEST) > 0) &
     CALL MPI_SEND(C, N**2, MPI_REAL, 0, MP_RANK, MP_LIBRARY_WORLD, IERROR)
     EXIT BLOCK
ND DO BLOCK

 There may be a matrix to receive from the "best" node.
     IF(MPI_NODE_PRIORITY(BEST) > 0 .and. MP_RANK == 0) &
```

```
       CALL MPI_RECV (C, N**2, MPI_REAL, MPI_ANY_SOURCE, MPI_ANY_TAG, &
         MP_LIBRARY_WORLD, STATUS, IERROR)

      if (count(int(C-A) /= 0) == 0 .and. MP_RANK == 0) &
        write (*,*) 'Parallel Example 16 is correct.'

 See to any error messages and exit MPI.
     mp_nprocs = mp_setup('Final')
     end
```

Parallel Example 17

Occasionally it is necessary to print output from all nodes of a communicator. This example has each non-root node prepare the output it will print in a character buffer. Then, each node in turn, the character buffer is transmitted to the root. The root prints the buffer, line-by-line, which contains an indication of where the output originated. Note that the root directs the order of results by broadcasting an integer value (BATON) giving the index of the node to transmit. The random numbers generated at the nodes and then listed are not checked. There is a final printed line indicating that the example is completed.

```
use show_int
use rand_int
       use mpi_setup_int

implicit none
       INCLUDE 'mpif.h'

 This is Parallel Example 17.  Each non-root node transmits
 the contents of an array that is the output of SHOW.
 The root receives the characters and prints the lines from
 alternate nodes.
       integer, parameter :: n=7, BSIZE=(72+2)*4
       integer k, p, q, ierror, status(MPI_STATUS_SIZE)
       integer I, BATON
       real(kind(1e0)) s_x(-1:n)
       type (s_options) options(7)
       CHARACTER (LEN=BSIZE) BUFFER
       character (LEN=12) PROC_NUM

 Setup for MPI:
       mp_nprocs = mp_setup()
f (mp_rank > 0) then
 The data types printed are real(kind(1e0)) random numbers.
s_x=rand(s_x)

 Convert node rank to CHARACTER data.
       write(proc_num,'(I3)') mp_rank

 Show 7 digits per number and  according to the
 natural or declared size of the array.
 Prepare the output lines in array BUFFER.
 End each line with ASCII sequence CR-NL.
       options(1)=show_significant_digits_is_7

       options(2)=show_starting_index_is
       options(3)= -1 ! The starting  value.

       options(4)=show_end_of_line_sequence_is
       options(5)=  2 ! Use 2 EOL characters.
       options(6)= 10 ! The ASCII code for CR.
       options(7)= 13 ! The ASCII code for NL.

       BUFFER= ' '    ! Blank out the buffer.

 Prepare the output in BUFFER.
call show (s_x, &
  'Rank-1, REAL with 7 digits, natural indexing from rank # '//&
  trim(adjustl(PROC_NUM)), IMAGE=BUFFER,  IOPT=options)

do i=1,mp_nprocs-1
 A handle or baton is received by the non-root nodes.
   call mpi_bcast(BATON, 1, MPI_INTEGER, 0, &
     MP_LIBRARY_WORLD, ierror)

 If this node has the baton, it transmits its buffer.
   if(BATON == mp_rank)&
     call mpi_send(buffer, BSIZE, MPI_CHARACTER, 0, mp_rank, &
       MP_LIBRARY_WORLD, ierror)
end do

lse
   DO I=1,MP_NPROCS-1

 The root sends out a handle to a node.  It is received as
 the value BATON.
     call mpi_bcast(I, 1, MPI_INTEGER, 0, &
       MP_LIBRARY_WORLD, ierror)

 A buffer of data arrives from a node.
     call mpi_recv(buffer, BSIZE, MPI_CHARACTER, MPI_ANY_SOURCE, &
       MPI_ANY_TAG, MP_LIBRARY_WORLD, STATUS, IERROR)

 Display BUFFER as a CHARACTER array. Discard blanks
 on the ends.  Look for non-printable characters as limits.
       p=0
       k=LEN(TRIM(BUFFER))
       DISPLAY:DO
         DO
           IF (p >= k) EXIT DISPLAY
           p=p+1
           IF(ICHAR(BUFFER(p:p)) >= ICHAR(' ')) EXIT
         END DO
         q=p-1
         DO
           q=q+1
           IF (ICHAR(BUFFER(q:q)) < ICHAR(' ')) EXIT
         END DO
         WRITE(*,'(1x,A)') BUFFER(p:q-1)
         p=q
       END DO DISPLAY
   END DO
nd if
 IF(MP_RANK ==0 ) &
   write(*,*) 'Parallel Example 17 is finished.'

 See to any error messages and quit MPI
   mp_nprocs = mp_setup('Final')

       end
```

Parallel Example 18

Here we illustrate a surface fitting problem implemented using tensor product B-splines with constraints. There are three functions, each depending on two parametric variables, for the spatial coordinates. Fitting each coordinate function to the data is a natural example of parallel computing in the sense that there are three separate problems of the same type. The approach is to break the problem into three data fitting computations. Each of these computations are allocated to nodes. Note that the data is sent from the root to the nodes.

Every node completes the least-squares fitting, and sends the spline coefficients back to the root node. This example requires four nodes to execute.

```
     USE surface_fitting_int
     USE rand_int
     USE norm_int
     USE Numerical_Libraries, only : DCONST
     USE mpi_setup_int
     implicit none

     INCLUDE 'mpif.h'

 This is a Parallel Example 18 for SURFACE_FITTING, or
 tensor product B-splines approximation.  Fit x, y, z parametric
 functions for points on the surface of a sphere of radius "A".
 Random values of latitude and longitude are used to generate
 data.  The functions are evaluated at a rectangular grid
 in latitude and longitude and checked so they lie on the
 surface of the sphere.

     integer :: i, j, ierror, status(MPI_STATUS_SIZE)
     integer, parameter :: ngrid=5, nord=8, ndegree=nord-1, &
       nbkpt=ngrid+2*ndegree, ndata =400, nvalues=50, NOPT=4
     real(kind(1d0)), parameter :: zero=0d0, one=1d0, two=2d0
     real(kind(1d0)), parameter :: TOLERANCE=1d-3
     real(kind(1d0)), target :: spline_data (4, ndata, 3), bkpt(nbkpt), &
        coeff(ngrid+ndegree-1,ngrid+ndegree-1, 3), delta, sizev, &
        pi, A, x(nvalues), y(nvalues), values(nvalues, nvalues), &
        data(4,ndata)

     real(kind(1d0)), pointer :: pointer_bkpt(:)
     type (d_surface_constraints), allocatable :: C(:)
     type (d_spline_knots) knotsx, knotsy
     type (d_options) OPTIONS(NOPT)

 Setup for MPI:
     MP_NPROCS = MP_SETUP()
LOCK: DO
 This program needs at least three nodes plus a root to execute.
 As many as three error messages may print.
     if(mp_nprocs < 4) then
       call e1sti (1, MP_NPROCS)
       call e1mes (5, 1, "Parallel Example 18 requires FOUR nodes"//&
         ' to execute. Number of nodes is now %(I1).')
       EXIT BLOCK
     endif

 Get the constant "pi" and a random radius, > 1.
     pi = DCONST((/'pi'/)); A=one+rand(A)

 Generate random (latitude, longitude) pairs and evaluate the
 surface parameters at these points.
     spline_data(1:2,:,1)=pi*(two*rand(spline_data(1:2,:,1))-one)
     spline_data(1:2,:,2)=spline_data(1:2,:,1)
     spline_data(1:2,:,3)=spline_data(1:2,:,1)

 Evaluate x, y, z parametric points.
     spline_data(3,:,1)=A*cos(spline_data(1,:,1))*cos(spline_data(2,:,1))
     spline_data(3,:,2)=A*cos(spline_data(1,:,2))*sin(spline_data(2,:,2))
     spline_data(3,:,3)=A*sin(spline_data(1,:,3))

 The values are equally uncertain.
     spline_data(4,:,:)=one

 Define the knots for the tensor product data fitting problem.
        delta = two*pi/(ngrid-1)
        bkpt(1:ndegree) = -pi
        bkpt(nbkpt-ndegree+1:nbkpt) =  pi
        bkpt(nord:nbkpt-ndegree)=(/(-pi+i*delta,i=0,ngrid-1)/)

 Assign the degree of the polynomial and the knots.
     pointer_bkpt => bkpt
     knotsx=d_spline_knots(ndegree, pointer_bkpt)
     knotsy=knotsx

 Fit a data surface for each coordinate.
 Set default regularization parameters to zero and compute
 residuals of the individual points. These are returned
 in DATA(4,:).
     allocate (C(2*ngrid))
 "Sew" the ends of the parametric surfaces together:
     do i=0,ngrid-1
       C(i+1)=surface_constraints(point=(/-pi,-pi+i*delta/),&
         type='.=.', periodic=(/pi,-pi+i*delta/))
     end do
     do i=0,ngrid-1
       C(ngrid+i+1)=surface_constraints(point=(/-pi+i*delta,-pi/),&
         type='.=.', periodic=(/-pi+i*delta,pi/))
     end do

     if (mp_rank == 0) then
 Send the data to a node.
        do j=1,3
          call mpi_send(spline_data(:,:,j), 4*ndata, &
           MPI_DOUBLE_PRECISION, j, j, MP_LIBRARY_WORLD, ierror)
        enddo
        do i=1,3
 Receive the coefficients back.
   call mpi_recv(coeff(:,:,i), (ngrid+ndegree-1)**2, &
            MPI_DOUBLE_PRECISION, i, i, MP_LIBRARY_WORLD, &
            status, ierror)
        enddo
     else if (mp_rank < 4) then

 Receive the data from the root.
       call mpi_recv(data, 4*ndata, MPI_DOUBLE_PRECISION, 0, &
         mp_rank, MP_LIBRARY_WORLD, status, ierror)
       OPTIONS(1)=d_options(surface_fitting_thinness,zero)
       OPTIONS(2)=d_options(surface_fitting_flatness,zero)
       OPTIONS(3)=d_options(surface_fitting_smallness,zero)
       OPTIONS(4)=surface_fitting_residuals

 Compute the coefficients at this node.
       coeff(:,:,mp_rank) = surface_fitting(data, knotsx, knotsy,&
         CONSTRAINTS=C, IOPT=OPTIONS)

 Send the coefficients back to the root.
call mpi_send(coeff(:,:,mp_rank),(ngrid+ndegree-1)**2,&
         MPI_DOUBLE_PRECISION, 0, mp_rank, MP_LIBRARY_WORLD,IERROR)
     end if

 Evaluate the function at a grid of points inside the rectangle of
 latitude and longitude covering the sphere just once.  Add the
 sum of squares. They should equal "A**2" but will not due to
 truncation and rounding errors.
     delta=pi/(nvalues+1)
     x=(/(-pi/two+i*delta,i=1,nvalues)/); y=two*x
     values=zero
     do j=1,3
       values=values + surface_values((/0,0/), x, y, knotsx, knotsy,&
         coeff(:,:,j))**2
     end do
     values=values-A**2

 Compute the R.M.S. error:
     sizev=norm(pack(values, (values == values)))/nvalues
     if (sizev <= TOLERANCE) then
       if(mp_rank == 0) &
       write(*,*) "Parallel Example 18 is correct."
     end if
    EXIT BLOCK
ND DO BLOCK

 See to any error messages and exit MPI.
     mp_nprocs = mp_setup('Final')
     end
```

Chapter 11: Utilities

Routines

Usage Notes for ScaLAPACK Utilities

MPI REQUIRED

This section describes the use of *ScaLAPACK,* a suite of dense linear algebra solvers, applicable when a single problem size is large. We have integrated usage of IMSL Fortran Library with *ScaLAPACK*. However, the *ScaLAPACK* library, including libraries for *BLACS* and *PBLAS*, are not part of this Library. To use *ScaLAPACK* software, the required libraries must be installed on the user's computer system. We adhered to the specification of Blackford, et al. (1997), but use only MPI for communication. The *ScaLAPACK* library includes certain *LAPACK* routines, Anderson, et al. (1995), redesigned for distributed memory parallel computers. It is written in a Single Program, Multiple Data (SPMD) style using explicit message passing for communication. Matrices are laid out in a two-dimensional block-cyclic decomposition. Using High Performance Fortran (HPF) directives, Koelbel, et al. (1994), and a *static* $p \times q$ processor array, and following declaration of the array, `A(*,*)`, this is illustrated by:

```
INTEGER, PARAMETER :: N=500, P= 2, Q=3, MB=32, NB=32
!HPF$ PROCESSORS PROC(P,Q)
!HPF$ DISTRIBUTE A(cyclic(MB), cyclic(NB)) ONTO PROC
```

Our integration work provides modules that describe the interface to the *ScaLAPACK* library. We recommend that users include these modules when using *ScaLAPACK* or ancillary packages, including *BLACS* and *PBLAS*. For the job of distributing data within a user's application to the block-cyclic decomposition required by *ScaLAPACK* solvers, we provide a utility that reads data from an external file and arranges the data within the distributed machines for a computational step. Another utility writes the results into an external file.

The data types supported for these utilities are integer; single precision, real; double precision, real; single precision, complex, and double precision, complex.

A *ScaLAPACK* library normally includes routines for:

- the solution of full-rank linear systems of equations,
- general and symmetric, positive-definite, banded linear systems of equations,
- general and symmetric, positive-definite, tri-diagonal, linear systems of equations,
- condition number estimation and iterative refinement for LU and Cholesky factorization,
- matrix inversion,

- full-rank linear least-squares problems,
- orthogonal and generalized orthogonal factorizations,
- orthogonal transformation routines,
- reductions to upper Hessenberg, bidiagonal and tridiagonal form,
- reduction of a symmetric-definite, generalized eigenproblem to standard form,
- the self-adjoint or Hermitian eigenproblem,
- the generalized self-adjoint or Hermitian eigenproblem, and
- the non-symmetric eigenproblem

ScaLAPACK routines are available in four data types: **single precision**, **real**; **double precision**; **real**, **single precision**, **complex**, and **double precision**, **complex**. At present, the non-symmetric eigenproblem is only available in single and double precision. More background information and user documentation is available on the World Wide Web at location *http://www.netlib.org/scalapack/slug/scalapack_slug.html*

- For users with rank deficiency or simple constraints in their linear systems or least-squares problem, we have routines for:

- full or deficient rank least-squares problems with non-negativity constraints
- full or deficient rank least-squares problems with simple upper and lower bound constraints

These are available in two data types: **single precision, real**, and **double precision, real**, and they are not part of *ScaLAPACK*. The matrices are distributed in a general block-column layout.

ScaLAPACK Supporting Modules

MPI REQUIRED

We recommend that users needing routines from *ScaLAPACK*, *PBLAS* or *BLACS*, Version 1.4, use modules that describe the interface to individual codes. This practice, including use of the declaration directive, `IMPLICIT NONE`, is a reliable way of writing *ScaLAPACK* application code, since the routines may have lengthy lists of arguments. Using the modules is helpful to avoid the mistakes such as missing arguments or mismatches involving Type, Kind or Rank (TKR). The modules are part of the Fortran Library product. There is a comprehensive module, `ScaLAPACK_Support`, that includes use of all the modules in the table below. This module decreases the number of lines of code for checking the interface, but at the cost of increasing source compilation time compared with using individual modules.

Module Name	Contents of the Module
ScaLAPACK_Support	All of the following modules
ScaLAPACK_Int	All interfaces to *ScaLAPACK* routines
PBLAS_Int	All interfaces to parallel *BLAS*, or *PBLAS*
BLACS_Int	All interfaces to basic linear algebra communication routines, or *BLACS*
TOOLS_Int LAPACK_Int	Interfaces to ancillary routines used by *ScaLAPACK*, but not in other packages All interfaces to *LAPACK* routines required by *ScaLAPACK*
ScaLAPACK_IO_Int	All interfaces to ScaLAPACK_Read, ScaLAPACK_Write utility routines. See this Chapter.
MPI_Node_Int	The module holding data describing the MPI communicator, MP_LIBRARY_WORLD. See Chapter 10.

ScaLAPACK_READ

MPI REQUIRED

This routine reads matrix data from a file and transmits it into the two-dimensional block-cyclic form required by *ScaLAPACK* routines. This routine contains a call to a barrier routine so that if one process is writing the file and an alternate process is to read it, the results will be synchronized.
All processors in the *BLACS* context call the routine.

Required Arguments

File_Name—(Input)
A character variable naming the file containing the matrix data. This file is opened with STATUS="OLD." If the name is misspelled or the file does not exist, or any access violation happens, a type = terminal error message will occur. After the contents are read, the file is closed. This file is read with a loop logically equivalent to groups of reads:

```
READ() ((BUFFER(I,J), I=1,M), J=1, NB)
```

or (optionally):

```
READ() ((BUFFER(I,J), J=1,N), I=1, MB)
```

DESC_A(*)—(Input)
The nine integer parameters associated with the *ScaLAPACK* matrix descriptor. Values for NB,MB,LDA are contained in this array.

A(LDA,*)—(Output)
This is an assumed-size array, with leading dimension LDA, that will contain this processor's piece of the block-cyclic matrix. The data type for A(*,*) is any of five Fortran intrinsic types, **integer, single precision, real; double precision, real; single precision, complex,** and **double precision-complex**.

Optional Arguments

`Format`—(Input)
A character variable containing a format to be used for reading the file containing matrix data. If this argument is not present, an unformatted, or list-directed read is used.

`iopt`—(Input)
Derived type array with the same precision as the array `A(*,*)`, used for passing optional data to `ScaLAPACK_READ`. The options are as follows:

Packaged Options for `ScaLAPACK_READ`		
Option Prefix = ?	Option Name	Option Value
`s_, d_`	`ScaLAPACK_READ_UNIT`	1
`s_, d_`	`ScaLAPACK_READ_FROM_PROCESS`	2
`s_, d_`	`ScaLAPACK_READ_BY_ROWS`	3

MPI REQUIRED

`iopt(IO) = ScaLAPACK_READ_UNIT`
Sets the unit number to the value in `iopt(IO + 1)%idummy`. The default unit number is the value 11.

`iopt(IO) = ScaLAPACK_READ_FROM_PROCESS`
Sets the process number that reads the named file to the value in `iopt(IO + 1)%idummy`. The default process number is the value 0.

`iopt(IO) = ScaLAPACK_READ_BY_ROWS`
Read the matrix by rows from the named file. By default the matrix is read by columns.

FORTRAN 90 Interface

Generic: `CALL ScaLAPACK_READ (File_Name, DESC_A, A [,…])`

Specific: The specific interface names are `S_ScaLAPACK_READ` and `D_ScaLAPACK_READ`.

Description

Subroutine `ScaLAPACK_READ` reads columns or rows of a problem matrix so that it is usable by a *ScaLAPACK* routine. It uses the two-dimensional block-cyclic array descriptor for the matrix to place the data in the desired assumed-size arrays on the processors. The blocks of data are read, then transmitted and received. The block sizes, contained in the array descriptor, determines the data set size for each blocking send and receive pair. The number of these synchronization points is proportional to $\lceil M \times N/(MB \times NB) \rceil$. A temporary local buffer is allocated for staging the matrix data. It is of size M by NB, when reading by columns, or N by MB, when reading by rows.

ScaLAPACK_WRITE

MPI REQUIRED

This routine writes the matrix data to a file. The data is transmitted from the two-dimensional block-cyclic form used by *ScaLAPACK*. This routine contains a call to a barrier routine so that if one process is writing the file
and an alternate process is to read it, the results will be synchronized. All processors in the *BLACS* context call the routine.

Required Arguments

File_Name—(Input)
A character variable naming the file to receive the matrix data. This file is opened with "STATUS="UNKNOWN." If any access violation happens, a type = terminal error message will occur. If the file already exists it will be overwritten. After the contents are written, the file is closed. This file is written with a loop logically equivalent to groups of writes:

```
WRITE() ((BUFFER(I,J), I=1,M), J=1, NB)
```

or (optionally):

```
WRITE() ((BUFFER(I,J), J=1,N), I=1, MB)
```

DESC_A(*)–(Input)
The nine integer parameters associated with the *ScaLAPACK* matrix descriptor. Values for NB,MB,LDA are contained in this array.

A(LDA,*) –(Input)
This is an assumed-size array, with leading dimension LDA, containing this processor's piece of the block-cyclic matrix. The data type for A(*,*) is any of five Fortran intrinsic types, **integer, single precision, real**, **double precision, real**, **single precision, complex**, and **double precision-complex**.

Optional Arguments

Format—(Input)
A character variable containing a format to be used for writing the file that receives matrix data. If this argument is not present, an unformatted, or list-directed write is used.

iopt—(Input)
Derived type array with the same precision as the array A(*,*), used for

passing optional data to `ScaLAPACK_WRITE`. Use single precision when `A(*,*)` is type `INTEGER`. The options are as follows:

Packaged Options for `ScaLAPACK_WRITE`		
Option Prefix = ?	Option Name	Option Value
`S_, d_`	`ScaLAPACK_WRITE_UNIT`	1
`S_, d_`	`ScaLAPACK_WRITE_FROM_PROCESS`	2
`S_, d_`	`ScaLAPACK_WRITE_BY_ROWS`	3

MPI REQUIRED

`iopt(IO) =ScaLAPACK_WRITE_UNIT`
Sets the unit number to the integer component of `iopt(IO + 1)%idummy`. The default unit number is the value 11.

`iopt(IO) = ScaLAPACK_WRITE_FROM_PROCESS`
Sets the process number that writes the named file to the integer component of `iopt(IO + 1)%idummy`. The default process number is the value 0.

`iopt(IO) = ScaLAPACK_WRITE_BY_ROWS`
Write the matrix by rows to the named file. By default the matrix is written by columns.

FORTRAN 90 Interface

Generic: `CALL ScaLAPACK_WRITE (File_Name, DESC_A, A [,…])`

Specific: The specific interface names are `S_ScaLAPACK_WRITE` and `D_ScaLAPACK_WRITE`.

Description

Subroutine `ScaLAPACK_WRITE` writes columns or rows of a problem matrix output by a *ScaLAPACK* routine. It uses the two-dimensional block-cyclic array descriptor for the matrix to extract the data from the assumed-size arrays on the processors. The blocks of data are transmitted and received, then written. The block sizes, contained in the array descriptor, determines the data set size for each blocking send and receive pair. The number of these synchronization points is proportional to $\lceil M \times N / (MB \times NB) \rceil$. A temporary local buffer is allocated for staging the matrix data. It is of size `M` by `NB`, when writing by columns, or `N` by `MB`, when writing by rows.

Example 1: Distributed Transpose of a Matrix, In Place

The program `SCPK_EX1` illustrates an *in-situ* transposition of a matrix. An $m \times n$ matrix, A, is written to a file, by rows. The $n \times m$ matrix, $B = A^T$, overwrites storage for A. Two temporary files are created and deleted. There is usage of the *BLACS* to define the process grid and provide further information identifying each process. This algorithm for transposing a matrix is not efficient. We use it to illustrate the read and write routines and optional arguments for writing of data by matrix rows.

```
  program scpk_ex1
! This is Example 1 for ScaLAPACK_READ and ScaLAPACK_WRITE.
! It shows in-situ or in-place transposition of a
! block-cyclic matrix.
USE ScaLAPACK_SUPPORT
USE ERROR_OPTION_PACKET
USE MPI_SETUP_INT

IMPLICIT NONE
INCLUDE "mpif.h"

INTEGER, PARAMETER :: M=6, N=6, MB=2, NB=2, NIN=10
INTEGER CONTXT, DESC_A(9), NPROW, NPCOL, MYROW, &
  MYCOL, IERROR, I, J, K, L, LDA, TDA
real(kind(1d0)), allocatable :: A(:,:), d_A(:,:)
real(kind(1d0)) ERROR
TYPE(d_OPTIONS) IOPT(1)
   MP_NPROCS=MP_SETUP()

   CALL BLACS_PINFO(MP_RANK, MP_NPROCS)
! Make initialization for BLACS.
   CALL BLACS_GET(0,0, CONTXT)

! Approximate processor grid to be nearly square.
   NPROW=sqrt(real(MP_NPROCS)); NPCOL=MP_NPROCS/NPROW
   IF(NPROW*NPCOL < MP_NPROCS) THEN
     NPROW=1; NPCOL=MP_NPROCS
   END IF
   CALL BLACS_GRIDINIT(CONTXT, 'Rows', NPROW, NPCOL)
! Get this processor's role in the process grid.
   CALL BLACS_GRIDINFO(CONTXT, NPROW, NPCOL, MYROW, MYCOL)
BLOCK: DO

LDA=NUMROC(M, MB, MYROW, 0, NPROW)
TDA=NUMROC(N, NB, MYCOL, 0, NPCOL)
  ALLOCATE(d_A(LDA,TDA))

! A root process is used to create the matrix data for the test.
IF(MP_RANK == 0) THEN
  ALLOCATE(A(M,N))
! Fill array with a pattern that is easy to recognize.
  K=0
  DO
   K=K+1; IF(10**K > N) EXIT
  END DO

  DO J=1,N
    DO I=1,M
! The values will appear, as decimals I.J, where I is
! the row and J is the column.
      A(I,J)=REAL(I)+REAL(J)*10d0**(-K)
    END DO
  END DO

  OPEN(UNIT=NIN, FILE='test.dat', STATUS='UNKNOWN')
! Write the data by columns.
  DO J=1,N,NB
    WRITE(NIN,*) ((A(I,L),I=1,M),L=J,min(N,J+NB-1))
  END DO
  CLOSE(NIN)
END IF

IF(MP_RANK == 0) THEN
  DEALLOCATE(A)
  ALLOCATE(A(N,M))
END IF

! Define the descriptor for the global matrix.
DESC_A=(/1, CONTXT, M, N, MB, NB, 0, 0, LDA/)

! Read the matrix into the local arrays.
CALL ScaLAPACK_READ('test.dat', DESC_A, d_A)

! To transpose, write the matrix by rows as the first step.
! This requires an option since the default is to write
! by columns.
IOPT(1)=ScaLAPACK_WRITE_BY_ROWS
CALL ScaLAPACK_WRITE("TEST.DAT", DESC_A, &
  d_A, IOPT=IOPT)

! Resize the local storage and read the transpose matrix.
  DEALLOCATE(d_A)
  LDA=NUMROC(N, MB, MYROW, 0, NPROW)
  TDA=NUMROC(M, NB, MYCOL, 0, NPCOL)
  ALLOCATE(d_A(LDA,TDA))

! Reshape the descriptor for the transpose of the matrix.
! The number of rows and columns are swapped.
DESC_A=(/1, CONTXT, N, M, MB, NB, 0, 0, LDA/)

CALL ScaLAPACK_READ("TEST.DAT", DESC_A, d_A)

IF(MP_RANK == 0) THEN

! Open the used files and delete when closed.
  OPEN(UNIT=NIN, FILE='test.dat', STATUS='OLD')
  CLOSE(NIN,STATUS='DELETE')
  OPEN(UNIT=NIN, FILE='TEST.DAT', STATUS='OLD')
  DO J=1,M,MB
    READ(NIN,*) ((A(I,L), I=1,N),L=J,min(M,J+MB-1))
  END DO
  CLOSE(NIN,STATUS='DELETE')
  DO I=1,N
    DO J=1,M
! The values will appear, as decimals I.J, where I is the row
!  and J is the column.
      A(I,J)=REAL(J)+REAL(I)*10d0**(-K) - A(I,J)
    END DO
  END DO
  ERROR=SUM(ABS(A))
 END IF

! The processors in use now exit the loop.
  EXIT BLOCK
END DO BLOCK

! See to any error messages.
  call e1pop("Mp_setup")

! Check results on just one process.
IF(ERROR <= SQRT(EPSILON(ERROR)) .and. &
  MP_RANK == 0) THEN
  write(*,*) " Example 1 for BLACS is correct."
END IF

! Deallocate storage arrays and exit from BLACS.
IF(ALLOCATED(A)) DEALLOCATE(A)
IF(ALLOCATED(d_A)) DEALLOCATE(d_A)

! Exit from using this process grid.
  CALL BLACS_GRIDEXIT( CONTXT )
  CALL BLACS_EXIT(0)
END
```

Output

```
Example 1 for BLACS is correct.
```

Example 2: Distributed Matrix Product with PBLAS

The program `SCPK_EX2` illustrates computation of the matrix product $C_{m\times n} = A_{m\times k}B_{k\times n}$. The matrices on the right-hand side are random. Three temporary files are created and deleted. There is usage of the *BLACS* and *PBLAS*. The problem sizes is such that the results are checked on one process.

```
  program scpk_ex2
! This is Example 2 for ScaLAPACK_READ and ScaLAPACK_WRITE.
! The product of two matrices is computed with PBLAS
! and checked for correctness.

USE ScaLAPACK_SUPPORT
USE MPI_SETUP_INT

IMPLICIT NONE
INCLUDE "mpif.h"

INTEGER, PARAMETER :: &
  K=32, M=33, N=34, MB=16, NB=16, NIN=10
INTEGER CONTXT, NPROW, NPCOL, MYROW, MYCOL, &
  INFO, IA, JA, IB, JB, IC, JC, LDA_A, TDA_A,&
  LDA_B, TDA_B, LDA_C, TDA_C, IERROR, I, J, L,&
  DESC_A(9), DESC_B(9), DESC_C(9)

real(kind(1d0)) :: ALPHA, BETA, ERROR=1d0, SIZE_C
real(kind(1d0)), allocatable, dimension(:,:) :: A,B,C,X(:),&
d_A, d_B, d_C

   MP_NPROCS=MP_SETUP()
! Routines with the "BLACS_" prefix are from the BLACS library.
! This is an adjunct library to the ScaLAPACK library.
   CALL BLACS_PINFO(MP_RANK, MP_NPROCS)

! Make initialization for BLACS.
   CALL BLACS_GET(0,0, CONTXT)

! Approximate processor grid to be nearly square.
   NPROW=sqrt(real(MP_NPROCS)); NPCOL=MP_NPROCS/NPROW
   IF(NPROW*NPCOL < MP_NPROCS) THEN
     NPROW=1; NPCOL=MP_NPROCS
```

```
       END IF
       CALL BLACS_GRIDINIT(CONTXT, 'Rows', NPROW, NPCOL)

    ! Get this processor's role in the process grid.
       CALL BLACS_GRIDINFO(CONTXT, NPROW, NPCOL, MYROW, MYCOL)

    ! Associate context (BLACS) with IMSL communicator:
       CALL BLACS_GET(CONTXT, 10, MP_LIBRARY_WORLD)

    BLOCK: DO

    ! Allocate local space for each array.
    LDA_A=NUMROC(M, MB, MYROW, 0, NPROW)
    TDA_A=NUMROC(K, NB, MYCOL, 0, NPCOL)
    LDA_B=NUMROC(K, NB, MYROW, 0, NPROW)
    TDA_B=NUMROC(N, NB, MYCOL, 0, NPCOL)
    LDA_C=NUMROC(M, MB, MYROW, 0, NPROW)
    TDA_C=NUMROC(N, NB, MYCOL, 0, NPCOL)

    ALLOCATE(d_A(LDA_A,TDA_A), d_B(LDA_B,TDA_B),&
      d_C(LDA_C,TDA_C))

    ! A root process is used to create the matrix data for the test.
    IF(MP_RANK == 0) THEN
      ALLOCATE(A(M,K), B(K,N), C(M,N), X(M))
      CALL RANDOM_NUMBER(A); CALL RANDOM_NUMBER(B)

      OPEN(UNIT=NIN, FILE='Atest.dat', STATUS='UNKNOWN')
    ! Write the data by columns.
      DO J=1,K,NB
        WRITE(NIN,*) ((A(I,L),I=1,M),L=J,min(K,J+NB-1))
      END DO
      CLOSE(NIN)

      OPEN(UNIT=NIN, FILE='Btest.dat', STATUS='UNKNOWN')
    ! Write the data by columns.
      DO J=1,N,NB
        WRITE(NIN,*) ((B(I,L),I=1,K),L=J,min(N,J+NB-1))
      END DO
      CLOSE(NIN)
    END IF

    ! Define the descriptor for the global matrices.
    DESC_A=(/1, CONTXT, M, K, MB, NB, 0, 0, LDA_A/)
    DESC_B=(/1, CONTXT, K, N, NB, NB, 0, 0, LDA_B/)
    DESC_C=(/1, CONTXT, M, N, MB, NB, 0, 0, LDA_C/)

    ! Read the factors into the local arrays.
    CALL ScaLAPACK_READ('Atest.dat', DESC_A, d_A)
    CALL ScaLAPACK_READ('Btest.dat', DESC_B, d_B)

    ! Compute the distributed product C = A x B.
    ALPHA=1d0; BETA=0d0
    IA=1; JA=1; IB=1; JB=1; IC=1; JC=1
    d_C=0
    CALL pdGEMM &
      ("No", "No", M, N, K, ALPHA, d_A, IA, JA,&
      DESC_A, d_B, IB, JB, DESC_B, BETA,&
      d_C, IC, JC, DESC_C )

    ! Put the product back on the root node.
    Call ScaLAPACK_WRITE('Ctest.dat', DESC_C, d_C)
```

```
IF(MP_RANK == 0) THEN

! Read the residuals and check them for size.
  OPEN(UNIT=NIN, FILE='Ctest.dat', STATUS='OLD')

! Read the data by columns.
  DO J=1,N,NB
    READ(NIN,*) ((C(I,L),I=1,M),L=J,min(N,J+NB-1))
  END DO

  CLOSE(NIN,STATUS='DELETE')
  SIZE_C=SUM(ABS(C)); C=C-matmul(A,B)
  ERROR=SUM(ABS(C))/SIZE_C

! Open other temporary files and delete them.
  OPEN(UNIT=NIN, FILE='Atest.dat', STATUS='OLD')
  CLOSE(NIN,STATUS='DELETE')
  OPEN(UNIT=NIN, FILE='Btest.dat', STATUS='OLD')
  CLOSE(NIN,STATUS='DELETE')

END IF

! The processors in use now exit the loop.
  EXIT BLOCK
END DO BLOCK

! See to any error messages.
   call e1pop("Mp_Setup")
! Deallocate storage arrays and exit from BLACS.
IF(ALLOCATED(A)) DEALLOCATE(A)
IF(ALLOCATED(B)) DEALLOCATE(B)
IF(ALLOCATED(C)) DEALLOCATE(C)
IF(ALLOCATED(X)) DEALLOCATE(X)
IF(ALLOCATED(d_A)) DEALLOCATE(d_A)
IF(ALLOCATED(d_B)) DEALLOCATE(d_B)
IF(ALLOCATED(d_C)) DEALLOCATE(d_C)

! Check the results.
IF(ERROR <= SQRT(EPSILON(ALPHA)) .and. &
  MP_RANK == 0) THEN
  write(*,*) " Example 2 for BLACS and PBLAS is correct."
END IF

! Exit from using this process grid.
  CALL BLACS_GRIDEXIT( CONTXT )
  CALL BLACS_EXIT(0)
END
```

Output

```
Example 2 for BLACS and PBLAS is correct.
```

Example 3: Distributed Linear Solver with ScaLAPACK

The program `SCPK_EX3` illustrates solving a system of linear-algebraic equations, $Ax = b$. The right-hand side is produced by defining A and y to have random values. Then the matrix-vector product $b = Ay$ is computed. The problem size is such that the residuals, $x - y \approx 0$ are checked on one process. Three temporary files are created and deleted. There is usage of the *BLACS* to define the process grid and provide further information identifying each process. Then *ScaLAPACK* is used to compute the approximate solution, x.

```
  program scpk_ex3
! This is Example 3 for ScaLAPACK_READ and ScaLAPACK_WRITE.
! A linear system is solved with ScaLAPACK and checked.
USE ScaLAPACK_SUPPORT
USE ERROR_OPTION_PACKET
USE MPI_SETUP_INT

IMPLICIT NONE

INCLUDE "mpif.h"
INTEGER, PARAMETER :: N=9, MB=3, NB=3, NIN=10
INTEGER CONTXT, NPROW, NPCOL, MYROW, MYCOL, &
  INFO, IA, JA, IB, JB, LDA_A, TDA_A,&
  LDA_B, TDA_B, IERROR, I, J, L, DESC_A(9),&
  DESC_B(9), DESC_X(9), BUFF(3), RBUF(3)

LOGICAL :: COMMUTE = .true.
INTEGER, ALLOCATABLE :: IPIV(:)
real(kind(1d0)) :: ERROR=0d0, SIZE_X
real(kind(1d0)), allocatable, dimension(:,:) :: A, B(:), &
  X(:), d_A, d_B

   MP_NPROCS=MP_SETUP()
! Routines with the "BLACS_" prefix are from the BLACS library.
   CALL BLACS_PINFO(MP_RANK, MP_NPROCS)
! Make initialization for BLACS.
   CALL BLACS_GET(0,0, CONTXT)

! Approximate processor grid to be nearly square.
   NPROW=sqrt(real(MP_NPROCS)); NPCOL=MP_NPROCS/NPROW
   IF(NPROW*NPCOL < MP_NPROCS) THEN
     NPROW=1; NPCOL=MP_NPROCS
   END IF
   CALL BLACS_GRIDINIT(CONTXT, 'Rows', NPROW, NPCOL)

! Get this processor's role in the process grid.
   CALL BLACS_GRIDINFO(CONTXT, NPROW, NPCOL, MYROW, MYCOL)

! Associate context (BLACS) with DNFL communicator:
   CALL BLACS_GET(CONTXT, 10, MP_LIBRARY_WORLD)

BLOCK: DO

! Allocate local space for each array.
LDA_A=NUMROC(N, MB, MYROW, 0, NPROW)
TDA_A=NUMROC(N, NB, MYCOL, 0, NPCOL)
LDA_B=NUMROC(N, MB, MYROW, 0, NPROW)
TDA_B=1
```

```
ALLOCATE(d_A(LDA_A,TDA_A), d_B(LDA_B,TDA_B),&
  IPIV(LDA_A+MB))

! A root process is used to create the matrix data for the test.
IF(MP_RANK == 0) THEN
  ALLOCATE(A(N,N), B(N), X(N))
  CALL RANDOM_NUMBER(A); CALL RANDOM_NUMBER(X)

! Compute the correct result.
  B=MATMUL(A,X); SIZE_X=SUM(ABS(X))
  OPEN(UNIT=NIN, FILE='Atest.dat', STATUS='UNKNOWN')

! Write the data by columns.
  DO J=1,N,NB
    WRITE(NIN,*) ((A(I,L),I=1,N),L=J,min(N,J+NB-1))
  END DO
  CLOSE(NIN)

  OPEN(UNIT=NIN, FILE='Btest.dat', STATUS='UNKNOWN')
! Write the data by columns.
  WRITE(NIN,*) (B(I),I=1,N)
  CLOSE(NIN)
END IF

! Define the descriptor for the global matrices.
DESC_A=(/1, CONTXT, N, N, MB, NB, 0, 0, LDA_A/)
DESC_B=(/1, CONTXT, N, 1, MB, NB, 0, 0, LDA_B/)
DESC_X=DESC_B

! Read the factors into the local arrays.
CALL ScaLAPACK_READ('Atest.dat', DESC_A, d_A)
CALL ScaLAPACK_READ('Btest.dat', DESC_B, d_B)

! Compute the distributed product solution to A x = b.
IA=1; JA=1; IB=1; JB=1

CALL pdGESV &
  (N, 1, d_A, IA, JA, DESC_A, IPIV, &
  d_B, IB, JB, DESC_B, INFO)


! Put the result on the root node.
Call ScaLAPACK_WRITE('Xtest.dat', DESC_B, d_B)

IF(MP_RANK == 0) THEN

! Read the residuals and check them for size.
  OPEN(UNIT=NIN, FILE='Xtest.dat', STATUS='OLD')

! Read the approximate solution data.
      READ(NIN,*) B
      B=B-X

  CLOSE(NIN,STATUS='DELETE')
  ERROR=SUM(ABS(B))/SIZE_X

! Delete temporary files.
  OPEN(UNIT=NIN, FILE='Atest.dat', STATUS='OLD')
  CLOSE(NIN,STATUS='DELETE')
  OPEN(UNIT=NIN, FILE='Btest.dat', STATUS='OLD')
  CLOSE(NIN,STATUS='DELETE')

END IF
```

```
! The processors in use now exit the loop.
  EXIT BLOCK
END DO BLOCK

! See to any error messages.
  call e1pop("Mp_Setup")

! Deallocate storage arrays and exit from BLACS.
IF(ALLOCATED(A)) DEALLOCATE(A)
IF(ALLOCATED(B)) DEALLOCATE(B)
IF(ALLOCATED(X)) DEALLOCATE(X)
IF(ALLOCATED(d_A)) DEALLOCATE(d_A)
IF(ALLOCATED(d_B)) DEALLOCATE(d_B)
IF(ALLOCATED(IPIV)) DEALLOCATE(IPIV)

IF(ERROR <= SQRT(EPSILON(ERROR)) .and.&
  MP_RANK == 0) THEN
  write(*,*) &
  " Example 3 for BLACS and ScaLAPACK solver is correct."
END IF

! Exit from using this process grid.
  CALL BLACS_GRIDEXIT( CONTXT )
  CALL BLACS_EXIT(0)
END
```

Output

```
Example 3 for BLACS and ScaLAPACK is correct.
```

ERROR_POST

Prints error messages that are generated by IMSL routines using EPACK.

Required Argument

EPACK — (Input [/Output])

Derived type array of size *p* containing the array of message numbers and associated data for the messages. The definition of this derived type is packaged within the modules used as interfaces for each suite of routines. The declaration is:

```
type ?_error
       integer idummy; real(kind(?_)) rdummy
end type
```

The choice of "`?_`" is either "`s_`" or "`d_`" depending on the accuracy of the data. This array gets additional messages and data from each routine that uses the "`epack=`" optional argument, provided *p* is large enough to hold data for a new message. The value *p* = 8 is sufficient to hold the longest single *terminal*, *fatal*, or *warning* message that an IMSL Fortran Library routine generates.

The location at entry `epack (1)%idummy` contains the number of data items for all messages. When the `error_post` routine exits, this value is set to zero. Locations in array positions `(2:) %idummy` contain groups of integers consisting of a message number, the *error severity* *level*, then the required integer data for the message. Floating-point data, if required in the message, is passed in locations(:)%rdummy matched with the starting point for integer data. The extent of the data for each message is determined by the requirements of the larger of each group of integer or floating-point values.

Optional Arguments

new_unit = nunit (Input)
Unit number, of type integer, associated for reading the direct-access file of error messages for the IMSL Fortran 90 routines.
Default: nunit = 4

new_path = path (Input)
Pathname in the local file space, of type character*64, needed for reading the direct-access file of error messages. Default string for path is defined during the installation procedure for certain IMSL Fortran Library routines.

FORTRAN 90 Interface

Generic: CALL ERROR_POST (EPACK [,…])

Specific: The specific interface names are S_ERROR_POST and D_ERROR_POST.

Description

A default direct-access error message file (.daf file) is supplied with this product. This file is read by error_post using the contents of the derived type argument epack, containing the message number, error severity level, and associated data. The message is converted into character strings accepted by the error processor and then printed. The number of pending messages that print depends on the settings of the parameters PRINT and STOP *IMSL MATH/LIBRARY User's Manual* (IMSL 1994, pp. 1194–1195). These values are initialized to defaults such that any *Level 5* or *Level 4* message causes a STOP within the error processor after a print of the text. To change these defaults so that more than one error message prints, use the routine ERSET documented and illustrated with examples in *IMSL MATH/LIBRARY User's Manual* (IMSL 1994, pp. 1196–1198). The method of using a message file to store the messages is required to support “shared-memory parallelism.”

Managing the Message File

For most applications of this product, there will be no need to manage this file. However, there are a few situations which may require changing or adding messages:

- New system-wide messages have been developed for applications using this Library.
- All or some of the existing messages need to be translated to another language
- A subset of users need to add a specific message file for their applications using this Library.

Following is information on changing the contents of the message file, and information on how to create and access a message file for a private application.

Changing Messages

In order to change messages, two files are required:

- An editable message glossary, messages.gls, supplied with this product.
- A source program, prepmess.f, used to generate an executable which builds messages.daf from messages.gls.

To change messages, first make a backup copy of `messages.gls`. Use a text editor to edit `messages.gls`. The format of this file is a series of pairs of statements:

- message_number=<nnnn>
- message='message string'

(Note that neither of these lines should begin with a tab.)

The variable `<nnnn>` is an integer message number (see below for ranges and reserved message numbers).

The `'message string'` is any valid message string not to exceed 255 characters. If a message line is too long for a screen, the standard Fortran 90 concatenation operator `//` with the line continuation character `&` may be used to wrap the text.

Most strings have substitution parameters embedded within them. These may be in the following forms:

- %(i<n>) for an integer substitution, where n is the nth integer output in this message.
- %(r<n>) for single precision real number substitution, where n is the nth real number output in this message.
- %(d<n>) for double precision real number substitution, where n is the nth double precision number output in this message.

New messages added to the system-wide error message file should be placed at the end of the file. Message numbers 5000 through 10000 have been reserved for user-added messages. Currently, messages 1 through 1400 are used by IMSL. Gaps in message number ranges are permitted; however, the message numbers must be in ascending order within the file. The message numbers used for each IMSL Fortran Library subroutine are documented in this manual and in online help.

If existing messages are being edited or translated, make sure not to alter the message_number lines. (This prevents conflicts with any new messages.gls file supplied with future versions of this Library.)

Building a New Direct-access Message File

The `prepmess` executable must be available to complete the message changing process. For information on building the `prepmess` executable from `prepmess.f` , consult the installation guide for this product.

Once new messages have been placed in the `messages.gls` file, make a backup copy of the `messages.daf` file. Then remove `messages.daf` from the current directory. Now enter the following command:

```
            prepmess > prepmess_output
```

A new messages.daf file is created. Edit the prepmess_output file and look near the end of the file for the new error messages. The prepmess program processes each message through the error message system as a validity check. There should be no FATAL error announcement within the prepmess_output file.

Private Message Files

Users can create a private message file within their own messages. This file would generally be used by an application that calls this Library. Follow the steps outlined above to created a private messages.gls file. The user should then be given a copy of the prepmess executable. In the application code, call the error_post subprogram with the new_unit/new_path optional arguments. The new path should point to the directory in which the private messages.daf file resides.

SHOW

Prints rank-1 or rank-2 arrays of numbers in a readable format.

Required Arguments

X — Rank-1 or rank-2 array containing the numbers to be printed. (Input)

Optional Arguments

text = CHARACTER (Input)
CHARACTER(LEN=*) string used for labeling the array.

image = buffer (Output)
CHARACTER(LEN=*) string used for an internal write buffer. With this argument present the output is converted to characters and packed. The lines are separated by an end-of-line sequence. The length of buffer is estimated by the line width in effect, time the number of lines for the array.

iopt = iopt(:) (Input)
Derived type array with the same precision as the input array; used for passing optional data to the routine. Use the REAL(KIND(1E0)) precision for output of INTEGER arrays. The options are as follows:

Packaged Options for SHOW		
Prefix is blank	Option Name	Option Value
	show_significant_digits_is_4	1
	show_significant_digits_is_7	2
	show_significant_digits_is_16	3
	show_line_width_is_44	4
	show_line_width_is_72	5
	show_line_width_is_128	6

Packaged Options for SHOW		
	show_end_of_line_sequence_is	7
	show_starting_index_is	8
	show_starting_row_index_is	9
	show_starting_col_index_is	10

```
iopt(IO) = show_significant_digits_is_4

iopt(IO) = show_significant_digits_is_7

iopt(IO) = show_significant_digits_is_16
```

These options allow more precision to be displayed. The default is 4D for each value. The other possible choices display 7D or 16D.

```
iopt(IO) = show_line_width_is_44

iopt(IO) = show_line_width_is_72

iopt(IO) = show_line_width_is_128
```

These options allow varying the output line width. The default is 72 characters per line. This allows output on many work stations or terminals to be read without wrapping of lines.

```
iopt(IO) = show_end-of_line_sequence_is
```

The sequence of characters ending a line when it is placed into the internal character buffer corresponding to the optional argument `'IMAGE = buffer'`. The value of `iopt(IO+1)%idummy` is the number of characters. These are followed, starting at `iopt(IO+2)%idummy`, by the *ASCII* codes of the characters themselves. The default is the single character, *ASCII* value 10 or *New Line*.

```
iopt(IO) = show_starting_index_is
```

This are used to reset the starting index for a rank-1 array to a value different from the default value, which is 1.

```
iopt(IO) = show_starting_row_index_is

iopt(IO) = show_starting_col_index_is
```

These are used to reset the starting row and column indices to values different from their defaults, each 1.

FORTRAN 90 Interface

Generic: `CALL SHOW (X [,…])`

Specific: The specific interface names are `S_SHOW` and `D_SHOW`.

Example 1: Printing an Array

Array of random numbers for all the intrinsic data types are printed. For `REAL(KIND(1E0))` rank-1 arrays, the number of displayed digits is reset from the default value of 4 to the value 7 and the subscripts for the array are reset so they match their declared extent when printed. The output is not shown.

```
       use show_int
       use rand_int

       implicit none

! This is Example 1 for SHOW.

       integer, parameter :: n=7, m=3
       real(kind(1e0)) s_x(-1:n), s_m(m,n)
       real(kind(1d0)) d_x(n), d_m(m,n)
       complex(kind(1e0)) c_x(n), c_m(m,n)
       complex(kind(1d0)) z_x(n),z_m(m,n)
       integer i_x(n), i_m(m,n)
        type (s_options) options(3)

! The data types printed are real(kind(1e0)), real(kind(1d0)), complex(kind(1e0)),
!complex(kind(1d0)), and INTEGER. Fill with randsom numbers
! and then print the contents, in each case with a label.
       s_x=rand(s_x); s_m=rand(s_m)
       d_x=rand(d_x); d_m=rand(d_m)
       c_x=rand(c_x); c_m=rand(c_m)
       z_x=rand(z_x); z_m=rand(z_m)
       i_x=100*rand(s_x(1:n)); i_m=100*rand(s_m)

       call show (s_x, 'Rank-1, REAL')
       call show (s_m, 'Rank-2, REAL')
       call show (d_x, 'Rank-1, DOUBLE')
       call show (d_m, 'Rank-2, DOUBLE')
       call show (c_x, 'Rank-1, COMPLEX')
       call show (c_m, 'Rank-2, COMPLEX')
       call show (z_x, 'Rank-1, DOUBLE COMPLEX')
       call show (z_m, 'Rank-2, DOUBLE COMPLEX')
       call show (i_x, 'Rank-1, INTEGER')
       call show (i_m, 'Rank-2, INTEGER')

! Show 7 digits per number and  according to the
! natural or declared size of the array.
        options(1)=show_significant_digits_is_7
        options(2)=show_starting_index_is
        options(3)= -1 ! The starting  value.
        call show (s_x, &
'Rank-1, REAL with 7 digits, natural indexing', IOPT=options)
        end
```

Output

```
Example 1 for SHOW is correct.
```

Description

The show routine is a generic subroutine interface to separate low-level subroutines for each data type and array shape. Output is directed to the unit number IUNIT. That number is obtained with the subroutine UMACH, *IMSL MATH/LIBRARY User's Manual* (IMSL 1994, pp. 1204–1205. Thus the user must open this unit in the calling program if it desired to be different from the standard output unit. If the optional argument ‘IMAGE = buffer‘ is present, the output is not sent to a file but to a character string within buffer. These characters are available to output or be used in the application.

Additional Examples

Example 2: Writing an Array to a Character Variable

This example prepares a rank-1 array for further processing, in this case delayed writing to the standard output unit. The indices and the amount of precision are reset from their defaults, as in Example 1. An end-of-line sequence of the characters CR-NL (*ASCII* 10,13) is used in place of the standard *ASCII* 10. This is not required for writing this array, but is included for an illustration of the option.

```
       use show_int
       use rand_int

       implicit none

! This is Example 2 for SHOW.
       integer, parameter :: n=7
       real(kind(1e0)) s_x(-1:n)
        type (s_options) options(7)
        CHARACTER (LEN=(72+2)*4) BUFFER
! The data types printed are real(kind(1e0)) random numbers.
       s_x=rand(s_x)


! Show 7 digits per number and  according to the
! natural or declared size of the array.
! Prepare the output lines in array BUFFER.
! End each line with ASCII sequence CR-NL.
        options(1)=show_significant_digits_is_7

        options(2)=show_starting_index_is
        options(3)= -1 ! The starting  value.

        options(4)=show_end_of_line_sequence_is
        options(5)=  2 ! Use 2 EOL characters.
        options(6)= 10 ! The ASCII code for CR.
        options(7)= 13 ! The ASCII code for NL.

        BUFFER= ' '    ! Blank out the buffer.

! Prepare the output in BUFFER.
 call show (s_x, &
 'Rank-1, REAL with 7 digits, natural indexing '//&
 'internal BUFFER, CR-NL EOLs.',&
 IMAGE=BUFFER,  IOPT=options)

! Display BUFFER as a CHARACTER array. Discard blanks
! on the ends.
        WRITE(*,'(1x,A)') TRIM(BUFFER)

        end
```

Output

```
Example 2 for SHOW is correct.
```

Fatal and Terminal Error Messages

See the *messages.gls* file for error messages for `show`. These error messages are numbered 601–606; 611–617; 621–627; 631–636; 641–646.

WRRRN

Prints a real rectangular matrix with integer row and column labels.

Required Arguments

TITLE — Character string specifying the title. (Input)
`TITLE` set equal to a blank character(s) suppresses printing of the title. Use "% /" within the title to create a new line. Long titles are automatically wrapped.

A — `NRA` by `NCA` matrix to be printed. (Input)

Optional Arguments

NRA — Number of rows. (Input)
Default: `NRA` = size (`A`, 1).

NCA — Number of columns. (Input)
Default: `NCA` = size (`A`,2).

LDA — Leading dimension of `A` exactly as specified in the dimension statement in the calling program. (Input)
Default: `LDA` = size (`A`,1).

ITRING — Triangle option. (Input)
Default: `ITRING` = 0.

ITRING	**Action**
0	Full matrix is printed.
1	Upper triangle of A is printed, including the diagonal.
2	Upper triangle of A excluding the diagonal of A is printed.
−1	Lower triangle of A is printed, including the diagonal.
−2	Lower triangle of A excluding the diagonal of A is printed.

FORTRAN 90 Interface

Generic: CALL WRRRN (TITLE, A [,…])

Specific: The specific interface names are S_WRRRN and D_WRRRN for two dimensional arrays, and S_WRRRN1D and D_WRRRN1D for one dimensional arrays.

FORTRAN 77 Interface

Single: CALL WRRRN (TITLE, NRA, NCA, A, LDA, ITRING)

Double: The double precision name is DWRRRN.

Example

The following example prints all of a 3 × 4 matrix A where $a_{ij} = i + j/10$.

```
      USE WRRRN_INT

      INTEGER    ITRING, LDA, NCA, NRA
      PARAMETER  (ITRING=0, LDA=10, NCA=4, NRA=3)
!
      INTEGER    I, J
      REAL       A(LDA,NCA)
!
      DO 20  I=1, NRA
         DO 10  J=1, NCA
            A(I,J) = I + J*0.1
   10    CONTINUE
   20 CONTINUE
!                                  Write A matrix.
      CALL WRRRN ('A', A, NRA=NRA)
      END
```

Output

```
                 A
        1       2       3       4
1   1.100   1.200   1.300   1.400
```

```
2   2.100   2.200   2.300   2.400
3   3.100   3.200   3.300   3.400
```

Comments

1. A single D, E, or F format is chosen automatically in order to print 4 significant digits for the largest element of A in absolute value. Routine WROPT (page 1591) can be used to change the default format.

2. Horizontal centering, a method for printing large matrices, paging, printing a title on each page, and many other options can be selected by invoking WROPT.

3. A page width of 78 characters is used. Page width and page length can be reset by invoking PGOPT (page 1599).

4. Output is written to the unit specified by UMACH (see the Reference Material).

Description

Routine WRRRN prints a real rectangular matrix with the rows and columns labeled 1, 2, 3, and so on. WRRRN can restrict printing to the elements of the upper or lower triangles of matrices via the ITRING option. Generally, ITRING ≠ 0 is used with symmetric matrices.

In addition, one-dimensional arrays can be printed as column or row vectors. For a column vector, set NRA to the length of the array and set NCA = 1. For a row vector, set NRA = 1 and set NCA to the length of the array. In both cases, set LDA = NRA and set ITRING = 0.

WRRRL

Print a real rectangular matrix with a given format and labels.

Required Arguments

TITLE — Character string specifying the title. (Input)
TITLE set equal to a blank character(s) suppresses printing of the title.

A — NRA by NCA matrix to be printed. (Input)

RLABEL — CHARACTER * (*) vector of labels for rows of A. (Input)
If rows are to be numbered consecutively 1, 2, …, NRA, use RLABEL(1) = 'NUMBER'. If no row labels are desired, use RLABEL(1) = 'NONE'. Otherwise, RLABEL is a vector of length NRA containing the labels.

CLABEL — CHARACTER * (*) vector of labels for columns of A. (Input)
If columns are to be numbered consecutively 1, 2, …, NCA, use CLABEL(1) = 'NUMBER'. If no column labels are desired, use CLABEL(1) = 'NONE'. Otherwise, CLABEL(1) is the heading for the row labels, and either CLABEL(2) must be

'NUMBER' or 'NONE', or CLABEL must be a vector of length NCA + 1 with CLABEL(1 + *j*) containing the column heading for the *j*-th column.

Optional Arguments

NRA — Number of rows. (Input)
Default: NRA = size (A,1).

NCA — Number of columns. (Input)
Default: NCA = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement in the calling program. (Input)
Default: LDA = size (A,1).

ITRING — Triangle option. (Input)
Default: ITRING = 0.

ITRING	**Action**
0	Full matrix is printed.
1	Upper triangle of A is printed, including the diagonal.
2	Upper triangle of A excluding the diagonal of A is printed.
−1	Lower triangle of A is printed, including the diagonal.
−2	Lower triangle of A excluding the diagonal of A is printed.

FMT — Character string containing formats. (Input)
If FMT is set to a blank character(s), the format used is specified by WROPT (page 1591). Otherwise, FMT must contain exactly one set of parentheses and one or more edit descriptors. For example, FMT = '(F10.3)' specifies this F format for the entire matrix. FMT = '(2E10.3, 3F10.3)' specifies an E format for columns 1 and 2 and an F format for columns 3, 4 and 5. If the end of FMT is encountered and if some columns of the matrix remain, format control continues with the first format in FMT. Even though the matrix A is real, an I format can be used to print the integer part of matrix elements of A. The most useful formats are special formats, called the "V and W formats," that can be used to specify pretty formats automatically. Set FMT = '(V10.4)' if you want a single D, E, or F format selected automatically with field width 10 and with 4 significant digits. Set FMT = '(W10.4)' if you want a single D, E, F, or I format selected automatically with field width 10 and with 4 significant digits. While the V format prints trailing zeroes and a trailing decimal point, the W format does not. See Comment 4 for general descriptions of the V and W formats. FMT may contain only D, E, F, G, I, V, or W edit descriptors, e.g., the X descriptor is not allowed.
Default: FMT = ‘ ‘.

FORTRAN 90 Interface

Generic: CALL WRRRL (TITLE, A, RLABEL, CLABEL [,…])

Specific: The specific interface names are S_WRRRL and D_WRRRL for two dimensional arrays, and S_WRRRL1D and D_WRRRL1D for one dimensional arrays.

FORTRAN 77 Interface

Single: CALL WRRRL (TITLE, NRA, NCA, A, LDA, ITRING, FMT, RLABEL, CLABEL)

Double: The double precision name is DWRRRL.

Example

The following example prints all of a 3 × 4 matrix *A* where $a_{ij} = (i + j/10)10^{j-3}$.

```
      USE WRRRL_INT
      INTEGER    ITRING, LDA, NCA, NRA
      PARAMETER  (ITRING=0, LDA=10, NCA=4, NRA=3)
!
      INTEGER    I, J
      REAL       A(LDA,NCA)
      CHARACTER  CLABEL(5)*5, FMT*8, RLABEL(3)*5
!
      DATA FMT/'(W10.6)'/
      DATA CLABEL/'    ', 'Col 1', 'Col 2', 'Col 3', 'Col 4'/
      DATA RLABEL/'Row 1', 'Row 2', 'Row 3'/
!
      DO 20  I=1, NRA
         DO 10  J=1, NCA
            A(I,J) = (I+J*0.1)*10.0**(J-3)
   10    CONTINUE
   20 CONTINUE
!                                  Write A matrix.
      CALL WRRRL ('A', A, RLABEL, CLABEL, NRA=NRA, FMT=FMT)
      END
```

Output

```
                              A
            Col 1        Col 2        Col 3        Col 4
Row 1       0.011        0.120        1.300       14.000
Row 2       0.021        0.220        2.300       24.000
Row 3       0.031        0.320        3.300       34.000
```

Comments

1. Workspace may be explicitly provided, if desired, by use of W2RRL/DW2RRL. The reference is:

```
CALL W2RRL (TITLE, NRA, NCA, A, LDA, ITRING, FMT,
RLABEL, CLABEL, CHWK)
```

The additional argument is:

CHWK — `CHARACTER` * 10 work vector of length `NCA`. This workspace is referenced only if all three conditions indicated at the beginning of this comment are met. Otherwise, `CHWK` is not referenced and can be a `CHARACTER` * 10 vector of length one.

2. The output appears in the following form:

```
                         TITLE
CLABEL(1)      CLABEL(2)  CLABEL(3)  CLABEL(4)
RLABEL(1)      Xxxxx      Xxxxx      Xxxxx
RLABEL(2)      Xxxxx      Xxxxx      Xxxxx
```

3. Use “% /” within titles or labels to create a new line. Long titles or labels are automatically wrapped.

4. For printing numbers whose magnitudes are unknown, the `G` format in FORTRAN is useful; however, the decimal points will generally not be aligned when printing a column of numbers. The `V` and `W` formats are special formats used by this routine to select a `D`, `E`, `F`, or `I` format so that the decimal points will be aligned. The `V` and `W` formats are specified as *Vn.d* and *Wn.d*. Here, *n* is the field width and *d* is the number of significant digits generally printed. Valid values for *n* are 3, 4,…, 40. Valid values for *d* are 1, 2, …, *n* – 2. If `FMT` specifies one format and that format is a `V` or `W` format, all elements of the matrix `A` are examined to determine one FORTRAN format for printing. If `FMT` specifies more than one format, FORTRAN formats are generated separately from each `V` or `W` format.

5. A page width of 78 characters is used. Page width and page length can be reset by invoking `PGOPT` (page 1599).

6. Horizontal centering, method for printing large matrices, paging, method for printing NaN (not a number), printing a title on each page, and many other options can be selected by invoking `WROPT` (page 1591).

7. Output is written to the unit specified by `UMACH` (see Reference Material).

Description

Routine `WRRRL` prints a real rectangular matrix (stored in *A*) with row and column labels (specified by `RLABEL` and `CLABEL`, respectively) according to a given format (stored in `FMT`). `WRRRL` can restrict printing to the elements of upper or lower triangles of matrices via the `ITRING` option. Generally, `ITRING` ≠ 0 is used with symmetric matrices.

In addition, one-dimensional arrays can be printed as column or row vectors. For a column vector, set NRA to the length of the array and set NCA = 1. For a row vector, set NRA = 1 and set NCA to the length of the array. In both cases, set LDA = NRA, and set ITRING = 0.

WRIRN

Prints an integer rectangular matrix with integer row and column labels.

Required Arguments

TITLE — Character string specifying the title. (Input)
TITLE set equal to a blank character(s) suppresses printing of the title. Use "% /" within the title to create a new line. Long titles are automatically wrapped.

MAT — NRMAT by NCMAT matrix to be printed. (Input)

Optional Arguments

NRMAT — Number of rows. (Input)
Default: NRMAT = size (MAT,1).

NCMAT — Number of columns. (Input)
Default: NCMAT = size (MAT,2).

LDMAT — Leading dimension of MAT exactly as specified in the dimension statement in the calling program. (Input)
Default: LDMAT = size (MAT,1).

ITRING — Triangle option. (Input)
Default: ITRING = 0.

ITRING	**Action**
0	Full matrix is printed.
1	Upper triangle of MAT is printed, including the diagonal.
2	Upper triangle of MAT excluding the diagonal of MAT is printed.
−1	Lower triangle of MAT is printed, including the diagonal.
−2	Lower triangle of MAT excluding the diagonal of MAT is printed.

FORTRAN 90 Interface

Generic: `CALL WRIRN (TITLE, MAT [,…])`

Specific: The specific interface name is S_WRIRN.

FORTRAN 77 Interface

Single: CALL WRIRN (TITLE, NRMAT, NCMAT, MAT, LDMAT, ITRING)

Example

The following example prints all of a 3 × 4 matrix A = MAT where $a_{ij} = 10i + j$.

```
      USE WRIRN_INT
      INTEGER    ITRING, LDMAT, NCMAT, NRMAT
      PARAMETER  (ITRING=0, LDMAT=10, NCMAT=4, NRMAT=3)
!
      INTEGER    I, J, MAT(LDMAT,NCMAT)
!
      DO 20  I=1, NRMAT
         DO 10  J=1, NCMAT
            MAT(I,J) = I*10 + J
   10    CONTINUE
   20 CONTINUE
!                                  Write MAT matrix.
      CALL WRIRN ('MAT', MAT, NRMAT=NRMAT)
      END
```

Output

```
         MAT
     1    2    3    4
1   11   12   13   14
2   21   22   23   24
3   31   32   33   34
```

Comments

1. All the entries in MAT are printed using a single I format. The field width is determined by the largest absolute entry.

2. Horizontal centering, a method for printing large matrices, paging, printing a title on each page, and many other options can be selected by invoking WROPT (page 1591).

3. A page width of 78 characters is used. Page width and page length can be reset by invoking PGOPT (page 1599).

4. Output is written to the unit specified by UMACH (see Reference Material).

Description

Routine WRIRN prints an integer rectangular matrix with the rows and columns labeled 1, 2, 3, and so on. WRIRN can restrict printing to elements of the upper and lower triangles of matrices via the ITRING option. Generally, ITRING ≠ 0 is used with symmetric matrices.

In addition, one-dimensional arrays can be printed as column or row vectors. For a column vector, set NRMAT to the length of the array and set NCMAT = 1. For a row vector, set NRMAT = 1 and set NCMAT to the length of the array. In both cases, set LDMAT = NRMAT and set ITRING = 0:

WRIRL

Print an integer rectangular matrix with a given format and labels.

Required Arguments

TITLE — Character string specifying the title. (Input)
TITLE set equal to a blank character(s) suppresses printing of the title.

MAT — NRMAT by NCMAT matrix to be printed. (Input)

RLABEL — CHARACTER * (*) vector of labels for rows of MAT. (Input)
If rows are to be numbered consecutively 1, 2, …, NRMAT, use RLABEL(1) = 'NUMBER'. If no row labels are desired, use RLABEL(1) = 'NONE'. Otherwise, RLABEL is a vector of length NRMAT containing the labels.

CLABEL — CHARACTER * (*) vector of labels for columns of MAT. (Input)
If columns are to be numbered consecutively 1, 2, …, NCMAT, use CLABEL(1) = 'NUMBER'. If no column labels are desired, use CLABEL(1) = 'NONE'. Otherwise, CLABEL(1) is the heading for the row labels, and either CLABEL(2) must be 'NUMBER' or 'NONE', or CLABEL must be a vector of length

NCMAT + 1 with CLABEL(1 + *j*) containing the column heading for the *j*-th column.

Optional Arguments

NRMAT — Number of rows. (Input)
Default: NRMAT = size (MAT,1).

NCMAT — Number of columns. (Input)
Default: NCMAT = size (MAT,2).

LDMAT — Leading dimension of MAT exactly as specified in the dimension statement in the calling program. (Input)
Default: LDMAT = size (MAT,1).

ITRING — Triangle option. (Input)
Default: ITRING = 0.

ITRING	**Action**
0	Full matrix is printed.
1	Upper triangle of MAT is printed, including the diagonal.
2	Upper triangle of MAT excluding the diagonal of MAT is printed.
−1	Lower triangle of MAT is printed, including the diagonal.
−2	Lower triangle of MAT excluding the diagonal of MAT is printed.

FMT — Character string containing formats. (Input)
If FMT is set to a blank character(s), the format used is a single I format with field width determined by the largest absolute entry. Otherwise, FMT must contain exactly one set of parentheses and one or more I edit descriptors. For example, FMT = ’(I10)’ specifies this I format for the entire matrix. FMT = ’(2I10, 3I5)’ specifies an I10 format for columns 1 and 2 and an I5 format for columns 3, 4 and 5. If the end of FMT is encountered and if some columns of the matrix remain, format control continues with the first format in FMT. FMT may only contain the I edit descriptor, e.g., the X edit descriptor is not allowed.
Default: FMT = ‘ ‘.

FORTRAN 90 Interface

Generic: CALL WRIRL (TITLE, MAT, RLABEL, CLABEL [,…])

Specific: The specific interface name is S_WRIRL.

FORTRAN 77 Interface

Single: CALL WRIRL (TITLE, NRMAT, NCMAT, MAT, LDMAT, ITRING, FMT, RLABEL, CLABEL)

Example

The following example prints all of a 3 × 4 matrix A = MAT where $a_{ij} = 10i + j$.

```
      USE WRIRL_INT
      INTEGER    ITRING, LDMAT, NCMAT, NRMAT

      PARAMETER  (ITRING=0, LDMAT=10, NCMAT=4, NRMAT=3)
!
      INTEGER    I, J, MAT(LDMAT,NCMAT)
      CHARACTER  CLABEL(5)*5, FMT*8, RLABEL(3)*5
!
      DATA FMT/’(I2)’/
      DATA CLABEL/’     ’, ’Col 1’, ’Col 2’, ’Col 3’, ’Col 4’/
      DATA RLABEL/’Row 1’, ’Row 2’, ’Row 3’/
```

```
!
      DO 20  I=1, NRMAT
         DO 10  J=1, NCMAT
            MAT(I,J) = I*10 + J
   10    CONTINUE
   20 CONTINUE
!                                Write MAT matrix.
      CALL WRIRL ('MAT', MAT, RLABEL, CLABEL, NRMAT=NRMAT)
      END
```

Output

```
                     MAT
        Col 1   Col 2   Col 3   Col 4
Row 1     11      12      13      14
Row 2     21      22      23      24
Row 3     31      32      33      34
```

Comments

1. The output appears in the following form:

   TITLE

CLABEL(1)	CLABEL(2)	CALBEL(3)	CLABEL 4)
RLABEL(1)	Xxxxx	xxxxx	xxxxx
RLABEL(2)	Xxxxx	xxxxx	xxxxx

2. Use “% /” within titles or labels to create a new line. Long titles or labels are automatically wrapped.

3. A page width of 78 characters is used. Page width and page length can be reset by invoking PGOPT (page 1599).

4. Horizontal centering, a method for printing large matrices, paging, printing a title on each page, and many other options can be selected by invoking WROPT (page 1591).

5. Output is written to the unit specified by UMACH (see the Reference Material).

Description

Routine WRIRL prints an integer rectangular matrix (stored in MAT) with row and column labels (specified by RLABEL and CLABEL, respectively), according to a given format (stored in FMT). WRIRL can restrict printing to the elements of upper or lower triangles of matrices via the ITRING option. Generally, ITRING $\neq 0$ is used with symmetric matrices. In addition, one-dimensional arrays can be printed as column or row vectors. For a column vector, set NRMAT to the length of the array and set NCMAT = 1. For a row vector, set NRMAT = 1 and set NCMAT to the length of the array. In both cases, set LDMAT = NRMAT, and set ITRING = 0.

WRCRN

Prints a complex rectangular matrix with integer row and column labels.

Required Arguments

TITLE — Character string specifying the title. (Input)
`TITLE` set equal to a blank character(s) suppresses printing of the title. Use “% /” within the title to create a new line. Long titles are automatically wrapped.

A — Complex `NRA` by `NCA` matrix to be printed. (Input)

Optional Arguments

NRA — Number of rows. (Input)
Default: `NRA` = size (`A`, 1).

NCA — Number of columns. (Input)
Default: `NCA` = size (`A`,2).

LDA — Leading dimension of `A` exactly as specified in the dimension statement in the calling program. (Input)
Default: `LDA` = size (`A`, 1).

ITRING — Triangle option. (Input)
Default: `ITRING` = 0.

ITRING	**Action**
0	Full matrix is printed.
1	Upper triangle of `A` is printed, including the diagonal.
2	Upper triangle of `A` excluding the diagonal of `A` is printed.
−1	Lower triangle of `A` is printed, including the diagonal.
−2	Lower triangle of `A` excluding the diagonal of `A` is printed.

FORTRAN 90 Interface

Generic: `CALL WRCRN (TITLE, A [,…])`

Specific: The specific interface names are `S_WRCRN` and `D_WRCRN` for two dimensional arrays, and `S_WRCRN1D` and `D_WRCRN1D` for one dimensional arrays.

FORTRAN 77 Interface

Single: CALL WRCRN (TITLE, NRA, NCA, A, LDA, ITRING)

Double: The double precision name is DWRCRN.

Example

This example prints all of a 3 × 4 complex matrix A with elements

$$a_{mn} = m + ni, \text{ where } i = \sqrt{-1}$$

```
      USE WRCRN_INT
      INTEGER    ITRING, LDA, NCA, NRA
      PARAMETER  (ITRING=0, LDA=10, NCA=4, NRA=3)
!
      INTEGER    I, J
      COMPLEX    A(LDA,NCA), CMPLX
      INTRINSIC  CMPLX
!
      DO 20  I=1, NRA
         DO 10  J=1, NCA
            A(I,J) = CMPLX(I,J)
   10    CONTINUE
   20 CONTINUE
!                                  Write A matrix.
      CALL WRCRN ('A', A, NRA=NRA)
      END
```

Output

```
                                  A
                1                  2                  3                  4
1  ( 1.000, 1.000)  ( 1.000, 2.000)  ( 1.000, 3.000)  ( 1.000, 4.000)
2  ( 2.000, 1.000)  ( 2.000, 2.000)  ( 2.000, 3.000)  ( 2.000, 4.000)
3  ( 3.000, 1.000)  ( 3.000, 2.000)  ( 3.000, 3.000)  ( 3.000, 4.000)
```

Comments

1. A single D, E, or F format is chosen automatically in order to print 4 significant digits for the largest real or imaginary part in absolute value of all the complex numbers in A. Routine WROPT (page 1591) can be used to change the default format.

2. Horizontal centering, a method for printing large matrices, paging, method for printing NaN (not a number), and printing a title on each page can be selected by invoking WROPT.

3. A page width of 78 characters is used. Page width and page length can be reset by invoking subroutine PGOPT (page 1599).

4. Output is written to the unit specified by UMACH (see Reference Material).

Description

Routine WRCRN prints a complex rectangular matrix with the rows and columns labeled 1, 2, 3, and so on. WRCRN can restrict printing to the elements of the upper or lower triangles of matrices via the ITRING option. Generally, ITRING ≠ 0 is used with Hermitian matrices.

In addition, one-dimensional arrays can be printed as column or row vectors. For a column vector, set NRA to the length of the array, and set NCA = 1. For a row vector, set NRA = 1, and set NCA to the length of the array. In both cases, set LDA = NRA, and set ITRING = 0.

WRCRL

Prints a complex rectangular matrix with a given format and labels.

Required Arguments

TITLE — Character string specifying the title. (Input)
TITLE set equal to a blank character(s) suppresses printing of the title.

A — Complex NRA by NCA matrix to be printed. (Input)

RLABEL — CHARACTER * (*) vector of labels for rows of A. (Input)
If rows are to be numbered consecutively 1, 2, …, NRA, use RLABEL(1) = 'NUMBER'. If no row labels are desired, use RLABEL(1) = 'NONE'. Otherwise, RLABEL is a vector of length NRA containing the labels.

CLABEL — CHARACTER * (*) vector of labels for columns of A. (Input)
If columns are to be numbered consecutively 1, 2, …, NCA, use CLABEL(1) = 'NUMBER'. If no column labels are desired, use CLABEL(1) = 'NONE'. Otherwise, CLABEL(1) is the heading for the row labels, and either CLABEL(2) must be 'NUMBER' or 'NONE', or CLABEL must be a vector of length NCA + 1 with CLABEL(1 + *j*) containing the column heading for the *j*-th column.

Optional Arguments

NRA — Number of rows. (Input)
Default: NRA = size (A,1).

NCA — Number of columns. (Input)
Default: NCA = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement in the calling program. (Input)
Default: LDA = size (A, 1).

ITRING — Triangle option. (Input)
Default: ITRING = 0.

ITRING	Action
0	Full matrix is printed.
1	Upper triangle of A is printed, including the diagonal.
2	Upper triangle of A excluding the diagonal of A is printed.
−1	Lower triangle of A is printed, including the diagonal.
−2	Lower triangle of A excluding the diagonal of A is printed.

FMT — Character string containing formats. (Input)
If FMT is set to a blank character(s), the format used is specified by WROPT (page 1591). Otherwise, FMT must contain exactly one set of parentheses and one or more edit descriptors. Because a complex number consists of two parts (a real and an imaginary part), two edit descriptors are used for printing a single complex number. FMT = ′(E10.3, F10.3)′ specifies an E format for the real part and an F format for the imaginary part. FMT = ′(F10.3)′ uses an F format for both the real and imaginary parts. If the end of FMT is encountered and if all columns of the matrix have not been printed, format control continues with the first format in FMT. Even though the matrix A is complex, an I format can be used to print the integer parts of the real and imaginary components of each complex number. The most useful formats are special formats, called the "V and W formats," that can be used to specify pretty formats automatically. Set FMT = ′(V10.4)′ if you want a single D, E, or F format selected automatically with field width 10 and with 4 significant digits. Set FMT = ′(W10.4)′ if you want a single D, E, F, or I format selected automatically with field width 10 and with 4 significant digits. While the V format prints trailing zeroes and a trailing decimal point, the W format does not. See Comment 4 for general descriptions of the V and W formats. FMT may contain only D, E, F, G, I, V, or W edit descriptors, e.g., the X descriptor is not allowed.
Default: FMT = ‘ ‘.

FORTRAN 90 Interface

Generic: CALL WRCRL (TITLE, A, RLABEL, CLABEL [,…])

Specific: The specific interface names are S_WRCRL and D_WRCRL for two dimensional arrays, and S_WRCRL1D and D_WRCRL1D for one dimensional arrays.

FORTRAN 77 Interface

Single: CALL WRCRL (TITLE, NRA, NCA, A, LDA, ITRING, FMT, RLABEL, CLABEL)

Double: The double precision name is DWRCRL.

Example

The following example prints all of a 3 × 4 matrix A with elements

$$a_{mn} = \left(m + .123456\right) + ni, \text{ where } i = \sqrt{-1}$$

```
      USE WRCRL_INT
      INTEGER    ITRING, LDA, NCA, NRA
      PARAMETER  (ITRING=0, LDA=10, NCA=4, NRA=3)
!
      INTEGER    I, J
      COMPLEX    A(LDA,NCA), CMPLX
      CHARACTER  CLABEL(5)*5, FMT*8, RLABEL(3)*5
      INTRINSIC  CMPLX
!
      DATA FMT/’(W12.6)’/
      DATA CLABEL/’     ’, ’Col 1’, ’Col 2’, ’Col 3’, ’Col 4’/
      DATA RLABEL/’Row 1’, ’Row 2’, ’Row 3’/
!
      DO 20  I=1, NRA
         DO 10  J=1, NCA
            A(I,J) = CMPLX(I,J) + 0.123456
   10    CONTINUE
   20 CONTINUE
!                                  Write A matrix.
      CALL WRCRL (’A’, A, RLABEL, CLABEL, NRA=NRA, FMT=FMT)
      END
```

Output

```
                               A
                             Col 1                        Col 2
Row 1  (     1.12346,     1.00000)  (     1.12346,     2.00000)
Row 2  (     2.12346,     1.00000)  (     2.12346,     2.00000)
Row 3  (     3.12346,     1.00000)  (     3.12346,     2.00000)

                             Col 3                        Col 4
Row 1  (     1.12346,     3.00000)  (     1.12346,     4.00000)
Row 2  (     2.12346,     3.00000)  (     2.12346,     4.00000)
Row 3  (     3.12346,     3.00000)  (     3.12346,     4.00000)
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `W2CRL/DW2CRL`. The reference is:

   ```
   CALL W2CRL (TITLE, NRA, NCA, A, LDA, ITRING, FMT,
   RLABEL, CLABEL, CHWK)
   ```

   The additional argument is:

   CHWK — `CHARACTER` * 10 work vector of length 2 * `NCA`. This workspace is referenced only if all three conditions indicated at the beginning of this comment are met. Otherwise, `CHWK` is not referenced and can be a `CHARACTER` * 10 vector of length one.

2. The output appears in the following form:

```
                                      TITLE
   CLABEL(1)            CLABEL(2)      CLABEL(3)      CLABEL(4)
   RLABEL(1)            (xxxxx,xxxxx)  (xxxxx,xxxxx)  (xxxxx,xxxxx)
   RLABEL(2)            (xxxxx,xxxxx)  (xxxxx,xxxxx)  (xxxxx,xxxxx)
```

3. Use “% /” within titles or labels to create a new line. Long titles or labels are automatically wrapped.

4. For printing numbers whose magnitudes are unknown, the `G` format in FORTRAN is useful; however, the decimal points will generally not be aligned when printing a column of numbers. The `V` and `W` formats are special formats used by this routine to select a `D`, `E`, `F`, or `I` format so that the decimal points will be aligned. The `V` and `W` formats are specified as *Vn.d* and *Wn.d*. Here, *n* is the field width, and *d* is the number of significant digits generally printed. Valid values for *n* are 3, 4, …, 40. Valid values for *d* are 1, 2, …, $n - 2$. If `FMT` specifies one format and that format is a `V` or `W` format, all elements of the matrix `A` are examined to determine one FORTRAN format for printing. If `FMT` specifies more than one format, FORTRAN formats are generated separately from each `V` or `W` format.

5. A page width of 78 characters is used. Page width and page length can be reset by invoking `PGOPT` (page 1599).

6. Horizontal centering, a method for printing large matrices, paging, method for printing NaN (not a number), printing a title on each page, and may other options can be selected by invoking `WROPT` (page 1591).

7. Output is written to the unit specified by `UMACH` (see the Reference Material).

Description

Routine `WRCRL` prints a complex rectangular matrix (stored in *A*) with row and column labels (specified by `RLABEL` and `CLABEL`, respectively) according to a given format (stored in `FMT`). Routine `WRCRL` can restrict printing to the elements of upper or lower triangles of matrices via the `ITRING` option. Generally, the `ITRING` ≠ 0 is used with Hermitian matrices.

In addition, one-dimensional arrays can be printed as column or row vectors. For a column vector, set `NRA` to the length of the array, and set `NCA` = 1. For a row vector, set `NRA` = 1, and set `NCA` to the length of the array. In both cases, set `LDA` = `NRA`, and set `ITRING` = 0.

WROPT

Sets or retrieves an option for printing a matrix.

Required Arguments

IOPT — Indicator of option type. (Input)

IOPT	**Description of Option Type**
−1, 1	Horizontal centering or left justification of matrix to be printed
−2, 2	Method for printing large matrices
−3, 3	Paging
−4, 4	Method for printing NaN (not a number), and negative and positive machine infinity.
−5, 5	Title option
−6, 6	Default format for real and complex numbers
−7, 7	Spacing between columns
−8, 8	Maximum horizontal space reserved for row labels
−9, 9	Indentation of continuation lines for row labels
−10, 10	Hot zone option for determining line breaks for row labels
−11, 11	Maximum horizontal space reserved for column labels
−12, 12	Hot zone option for determining line breaks for column labels
−13, 13	Hot zone option for determining line breaks for titles
−14, 14	Option for the label that appears in the upper left hand corner that can be used as a heading for the row numbers or a label for the column headings for WR**N routines
−15, 15	Option for skipping a line between invocations of WR**N routines, provided a new page is not to be issued
−16, 16	Option for vertical alignment of the matrix values relative to the associated row labels that occupy more than one line
0	Reset all the current settings saved in internal variables back to their last setting made with an invocation of WROPT with ISCOPE = 1. (This option is used internally by routines printing a matrix and is not useful otherwise.)

If IOPT is negative, ISETNG and ISCOPE are input and are saved in internal variables. If IOPT is positive, ISETNG is output and receives the currently active setting for the option (if ISCOPE = 0) or the last global setting for the option (if ISCOPE = 1). If IOPT = 0, ISETNG and ISCOPE are not referenced.

ISETNG — Setting for option selected by IOPT. (Input, if IOPT is negative; output, if IOPT is positive; not referenced if IOPT = 0)

IOPT	ISETNG	Meaning
−1, 1	0	Matrix is left justified
	1	Matrix is centered horizontally on page
−2, 2	0	A complete row is printed before the next row is printed. Wrapping is used if necessary.
	M	Here, m is a positive integer. Let nı be the maximum number of columns beginning with column 1 that fit across the page (as determined by the widths of the printing formats). First, columns 1 through nı are printed for rows 1 through m. Let n□ be the maximum number of columns beginning with column nı + 1 that fit across the page. Second, columns nı + 1 through nı + n□ are printed for rows 1 through m. This continues until the last columns are printed for rows 1 through m. Printing continues in this fashion for the next m rows, etc.
−3, 3	−2	Printing begins on the next line, and no paging occurs.
	−1	Paging is on. Every invocation of a WR*** routine begins on a new page, and paging occurs within each invocation as is needed
	0	Paging is on. The first invocation of a WR*** routine begins on a new page, and subsequent paging occurs as is needed. With this option, every invocation of a WR*** routine ends with a call to WROPT to reset this option to k, a positive integer giving the number of lines printed on the current page.

	K	Here, k is a positive integer. Paging is on, and k lines have been printed on the current page. If k is less than the page length IPAGE (see PGOPT, page 1599), then IPAGE − k lines are printed before a new page instruction is issued. If k is greater than or equal to IPAGE, then the first invocation of a WR*** routine begins on a new page. In any case, subsequent paging occurs as is needed. With this option, every invocation of a WR*** routine ends with a call to WROPT to reset the value of k.
−4, 4	0	NaN is printed as a series of decimal points, negative machine infinity is printed as a series of minus signs, and positive machine infinity is printed as a series of plus signs.
	1	NaN is printed as a series of blank characters, negative machine infinity is printed as a series of minus signs, and positive machine infinity is printed as a series of plus signs.
	2	NaN is printed as “NaN,” negative machine infinity is printed as “-Inf” and positive machine infinity is printed as “Inf.”
	3	NaN is printed as a series of blank characters, negative machine infinity is printed as “-Inf,” and positive machine infinity is printed as “Inf.”
−5, 5	0	Title appears only on first page.
	1	Title appears on the first page and all continuation pages.
−6, 6	0	Format is (W10.4). See Comment 2.
	1	Format is (W12.6). See Comment 2.
	2	Format is (1PE12.5).
	3	Format is Vn.4 where the field width n is determined. See Comment 2.
	4	Format is *Vn*.6 where the field width *n* is determined. Comment 2.
	5	Format is 1PEn.d where n = d + 7, and d + 1 is the maximum number of significant digits.
−7, 7	K_1	Number of characters left blank between columns. k_1 must be between 0 and 5, inclusively.
−8, 8	K^2	Maximum width (in characters) reserved for row labels. K^2 = 0 means use the default.

−9, 9	K^3	Number of characters used to indent continuation lines for row labels. k^3 must be between 0 and 10, inclusively.
−10, 10	K^4	Width (in characters) of the hot zone where line breaks in row labels can occur. $k^4 = 0$ means use the default. k^4 must not exceed 50.
−11, 11	K^5	Maximum width (in characters) reserved for column labels. k□ = 0 means use the default.
−12, 12	K^6	Width (in characters) of the hot zone where line breaks in column labels can occur. $k^6 = 0$ means use the default. k^6 must not exceed 50.
−13, 13	K^7	Width (in characters) of the hot zone where line breaks in titles can occur. k^7 must be between 1 and 50, inclusively.
−14	0	There is no label in the upper left hand corner.
	1	The label in the upper left hand corner is “Component” if a row vector or column vector is printed; the label is “Row/Column” if both the number of rows and columns are greater than one; otherwise, there is no label.
−15	0	A blank line is printed on each invocation of a WR**N routine before the matrix title provided a new page is not to be issued.
	1	A blank line is not printed on each invocation of a WR**N routine before the matrix title.
−16, 16	0	The matrix values are aligned vertically with the last line of the associated row label for the case IOPT = 2 and ISET is positive.
	1	The matrix values are aligned vertically with the first line of the associated row label.

ISCOPE — Indicator of the scope of the option. (Input if `IOPT` is nonzero; not referenced if `IOPT` = 0)

ISCOPE	**Action**
0	Setting is temporarily active for the next invocation of a `WR***` matrix printing routine.
1	Setting is active until it is changed by another invocation of `WROPT`.

FORTRAN 90 Interface

Generic: `CALL WROPT (IOPT, ISETNG, ISCOPE)`

Specific: The specific interface name is WROPT.

FORTRAN 77 Interface

Single: CALL WROPT (IOPT, ISETNG, ISCOPE)

Example

The following example illustrates the effect of WROPT when printing a 3 × 4 real matrix A with WRRRN (page 1553) where $a_{ij} = i + j/10$. The first call to WROPT sets horizontal printing so that the matrix is first printed horizontally centered on the page. In the next invocation of WRRRN, the left-justification option has been set via routine WROPT so the matrix is left justified when printed. Finally, because the scope of left justification was only for the next call to a printing routine, the last call to WRRRN results in horizontally centered printing.

```
      USE WROPT_INT
      USE WRRRN_INT
      INTEGER    ITRING, LDA, NCA, NRA
      PARAMETER  (ITRING=0, LDA=10, NCA=4, NRA=3)
!
      INTEGER    I, IOPT, ISCOPE, ISETNG, J
      REAL       A(LDA,NCA)
!
      DO 20  I=1, NRA
         DO 10  J=1, NCA
            A(I,J) = I + J*0.1
   10    CONTINUE
   20 CONTINUE
!                                 Activate centering option.
!                                 Scope is global.
      IOPT   = -1
      ISETNG = 1
      ISCOPE = 1
!
      CALL WROPT (IOPT, ISETNG, ISCOPE)
!                                 Write A matrix.
      CALL WRRRN ('A', A, NRA=NRA)
!                                 Activate left justification.
!                                 Scope is local.
      IOPT   = -1
      ISETNG   = 0
      ISCOPE = 0
      CALL WROPT (IOPT, ISETNG, ISCOPE)
      CALL WRRRN ('A', A, NRA=NRA)
      CALL WRRRN ('A', A, NRA=NRA)
      END
```

Output

```
                                   A
                     1         2         3         4
             1   1.100     1.200     1.300     1.400
             2   2.100     2.200     2.300     2.400
```

```
                       3   3.100   3.200   3.300   3.400

                A
        1       2       3       4
1   1.100   1.200   1.300   1.400
2   2.100   2.200   2.300   2.400
3   3.100   3.200   3.300   3.400

                                       A
                               1       2       3       4
                       1   1.100   1.200   1.300   1.400
                       2   2.100   2.200   2.300   2.400
                       3   3.100   3.200   3.300   3.400
```

Comments

1. This program can be invoked repeatedly before using a `WR***` routine to print a matrix. The matrix printing routines retrieve these settings to determine the printing options. It is not necessary to call `WROPT` if a default value of a printing option is desired. The defaults are as follows.

IOPT	Default Value for ISET	Meaning
1	0	Left justified
2	1000000	Number lines before wrapping
3	–2	No paging
4	2	NaN is printed as “NaN,” negative machine infinity is printed as “-Inf” and positive machine infinity is printed as “Inf.”
5	0	Title only on first page.
6	3	Default format is Vn.4.
7	2	2 spaces between columns.
8	0	Maximum row label width `MAXRLW` = 2 * `IPAGEW`/3 if matrix has one column; `MAXRLW` = `IPAGEW`/4 otherwise.
9	3	3 character indentation of row labels continued beyond one line.
10	0	Width of row label hot zone is `MAXRLW`/3 characters.

11	0	Maximum column label width MAXCLW = min{max (NW + NW/2, 15), 40} for integer and real matrices, where NW is the field width for the format corresponding to the particular column. MAXCLW = min{max(NW + NW/2, 15), 83} for complex matrices, where NW is the sum of the two field widths for the formats corresponding to the particular column plus 3.
12	0	Width of column label hot zone is MAXCLW/3 characters.
13	10	Width of hot zone for titles is 10 characters.
14	0	There is no label in the upper left hand corner.
15	0	Blank line is printed.
16	0	The matrix values are aligned vertically with the last line of the associated row label.

For IOPT = 8, the default depends on the current value for the page width, IPAGEW (see PGOPT, page 1599).

2. The V and W formats are special formats that can be used to select a D, E, F, or I format so that the decimal points will be aligned. The V and W formats are specified as *Vn.d* and *Wn.d*. Here, *n* is the field width and *d* is the number of significant digits generally printed. Valid values for *n* are 3, 4, …, 40. Valid values for *d* are 1, 2, …, *n* – 2. While the V format prints trailing zeroes and a trailing decimal point, the W format does not.

Description

Routine WROPT allows the user to set or retrieve an option for printing a matrix. The options controlled by WROPT include the following: horizontal centering, a method for printing large matrices, paging, method for printing NaN (not a number) and positive and negative machine infinities, printing titles, default formats for numbers, spacing between columns, maximum widths reserved for row and column labels, indentation of row labels that continue beyond one line, widths of hot zones for breaking of labels and titles, the default heading for row labels, whether to print a blank line between invocations of routines, and vertical alignment of matrix entries with respect to row labels continued beyond one line. (NaN and positive and negative machine infinities can be retrieved by AMACH and DMACH that are documented in the section “Machine-Dependent Constants” in the Reference Material.) Options can be set globally

(ISCOPE = 1) or temporarily for the next call to a printing routine (ISCOPE = 0).

PGOPT

Sets or retrieves page width and length for printing.

Required Arguments

IOPT — Page attribute option. (Input)

IOPT	**Description of Attribute**
−1, 1	Page width.
−2, 2	Page length.

Negative values of IOPT indicate the setting IPAGE is input. Positive values

of IOPT indicate the setting IPAGE is output.

IPAGE — Value of page attribute. (Input, if IOPT is negative; output, if IOPT is positive.)

IOPT	**Description of Attribute**	**Settings for IPAGE**
−1, 1	Page width (in characters)	10, 11, …
−2, 2	Page length (in lines)	10, 11, …

FORTRAN 90 Interface

Generic: CALL PGOPT (IOPT, IPAGE)

Specific: The specific interface name is PGOPT.

FORTRAN 77 Interface

Single: CALL PGOPT (IOPT, IPAGE)

Example

The following example illustrates the use of PGOPT to set the page width at 20 characters. Routine WRRRN (page 1553) is then used to print a 3 × 4 matrix A where $a_{ij}= i + j/10$.

```
      USE PGOPT_INT
      USE WRRRN_INT
      INTEGER    ITRING, LDA, NCA, NRA
      PARAMETER  (ITRING=0, LDA=3, NCA=4, NRA=3)
!
      INTEGER    I, IOPT, IPAGE, J
      REAL       A(LDA,NCA)
```

```
!
      DO 20  I=1, NRA
         DO 10  J=1, NCA
            A(I,J) = I + J*0.1
   10    CONTINUE
   20 CONTINUE
!                                  Set page width.
      IOPT  = -1
      IPAGE = 20
      CALL PGOPT (IOPT, IPAGE)
!                                  Print the matrix A.
      CALL WRRRN ('A', A)
      END
```

Output

```
         A
        1       2
1   1.100   1.200
2   2.100   2.200
3   3.100   3.200

        3       4
1   1.300   1.400
2   2.300   2.400
3   3.300   3.400
```

Description

Routine PGOPT is used to set or retrieve the page width or the page length for routines that perform printing.

PERMU

Rearranges the elements of an array as specified by a permutation.

Required Arguments

X — Real vector of length N containing the array to be permuted. (Input)

IPERMU — Integer vector of length N containing a permutation IPERMU(1), …, IPERMU(N) of the integers 1, …, N. (Input)

XPERMU — Real vector of length N containing the array X permuted. (Output)
If X is not needed, X and XPERMU can share the same storage locations.

Optional Arguments

N — Length of the arrays X and XPERMU. (Input)
Default: N = size (IPERMU,1).

IPATH — Integer flag. (Input)

Default: `IPATH` = 1.

`IPATH` = 1 means `IPERMU` represents a forward permutation, i.e., `X`(`IPERMU`(`I`)) is moved to `XPERMU`(`I`). `IPATH` = 2 means `IPERMU` represents a backward permutation, i.e., `X`(`I`) is moved to `XPERMU`(`IPERMU`(`I`)).

FORTRAN 90 Interface

Generic: `CALL PERMU` (`X`, `IPERMU`, `XPERMU` `[,…]`)

Specific: The specific interface names are `S_PERMU` and `D_PERMU`.

FORTRAN 77 Interface

Single: `CALL PERMU (N, X, IPERMU, IPATH, XPERMU)`

Double: The double precision name is `DPERMU`.

Example

This example rearranges the array *X* using `IPERMU`; forward permutation is performed.

```
      USE PERMU_INT
      USE UMACH_INT
!                                 Declare variables
      INTEGER    IPATH, N
      PARAMETER  (IPATH=1, N=4)
!
      INTEGER    IPERMU(N), J, NOUT
      REAL       X(N), XPERMU(N)
!                                 Set values for  X, IPERMU
!
!                           X = ( 5.0  6.0  1.0  4.0 )
!                           IPERMU = ( 3 1 4 2 )
!
      DATA X/5.0, 6.0, 1.0, 4.0/, IPERMU/3, 1, 4, 2/
!                                 Permute X into XPERMU
      CALL PERMU (X, IPERMU, XPERMU)
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Print results
      WRITE (NOUT,99999) (XPERMU(J),J=1,N)
!
99999 FORMAT ('  The output vector is:', /, 10(1X,F10.2))
      END
```

Output

```
The Output vector is:
1.00       5.00       4.00       6.00
```

Description

Routine PERMU rearranges the elements of an array according to a permutation vector. It has the option to do both forward and backward permutations.

PERMA

Permutes the rows or columns of a matrix.

Required Arguments

A — NRA by NCA matrix to be permuted. (Input)

IPERMU — Vector of length K containing a permutation IPERMU(1), …, IPERMU(K) of the integers 1, …, K where K = NRA if the rows of A are to be permuted and K = NCA if the columns of A are to be permuted. (Input)

APER — NRA by NCA matrix containing the permuted matrix. (Output)
If A is not needed, A and APER can share the same storage locations.

Optional Arguments

NRA — Number of rows. (Input)
Default: NRA = size (A,1).

NCA — Number of columns. (Input)
Default: NCA = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A, 1).

IPATH — Option parameter. (Input)
IPATH = 1 means the rows of A will be permuted. IPATH = 2 means the columns of A will be permuted.
Default: IPATH = 1.

LDAPER — Leading dimension of APER exactly as specified in the dimension statement of the calling program. (Input)
Default: LDAPER = size (APER,1).

FORTRAN 90 Interface

Generic: CALL PERMA (A, IPERMU, APER [,…])

Specific: The specific interface names are S_PERMA and D_PERMA.

FORTRAN 77 Interface

Single: `CALL PERMA (NRA, NCA, A, LDA, IPERMU, IPATH, APER, LDAPER)`

Double: The double precision name is `DPERMA`.

Example

This example permutes the columns of a matrix *A*.

```
      USE PERMA_INT
      USE UMACH_INT
!                                 Declare variables
      INTEGER    IPATH, LDA, LDAPER, NCA, NRA
      PARAMETER  (IPATH=2, LDA=3, LDAPER=3, NCA=5, NRA=3)
!
      INTEGER    I, IPERMU(5), J, NOUT
      REAL       A(LDA,NCA), APER(LDAPER,NCA)
!                                 Set values for  A, IPERMU
!                                 A = ( 3.0  5.0  1.0  2.0  4.0 )
!                                     ( 3.0  5.0  1.0  2.0  4.0 )
!                                     ( 3.0  5.0  1.0  2.0  4.0 )
!
!                                 IPERMU = ( 3 4 1 5 2 )
!
      DATA A/3*3.0, 3*5.0, 3*1.0, 3*2.0, 3*4.0/, IPERMU/3, 4, 1, 5, 2/
!                                 Perform column permutation on A,
!                                 giving APER
      CALL PERMA (A, IPERMU, APER, IPATH=IPATH)
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Print results
      WRITE (NOUT,99999) ((APER(I,J),J=1,NCA),I=1,NRA)
!
99999 FORMAT (’  The output matrix is:’, /, 3(5F8.1,/))
      END
```

Output

```
The Output matrix is:
1.0     2.0     3.0     4.0     5.0
1.0     2.0     3.0     4.0     5.0
1.0     2.0     3.0     4.0     5.0
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `P2RMA`/`DP2RMA`. The reference is:

   `CALL P2RMA (NRA, NCA, A, LDA, IPERMU, IPATH, APER, LDAPER, WORK)`

   The additional argument is:

   WORK — Real work vector of length `NCA`.

Description

Routine PERMA interchanges the rows or columns of a matrix using a permutation vector such as the one obtained from routines SVRBP (page 1614) or SVRGP (page 1608).

The routine PERMA permutes a column (row) at a time by calling PERMU (page 1600). This process is continued until all the columns (rows) are permuted. On completion, let B = APER and p_i = IPERMU(I), then

$$B_{ij} = A_{p_i j}$$

for all i, j.

SORT_REAL

Sorts a rank-1 array of real numbers x so the y results are algebraically nondecreasing, $y_1 \le y_2 \le \ldots y_n$.

Required Arguments

X — Rank-1 array containing the numbers to be sorted. (Output)

Y — Rank-1 array containing the sorted numbers. (Output)

Optional Arguments

NSIZE = n (Input)
Uses the sub-array of size n for the numbers.
Default value: n = size(x)

IPERM = iperm (Input/Output)
Applies interchanges of elements that occur to the entries of iperm(:). If the values iperm(i)=i,i=1,n are assigned prior to call, then the output array is moved to its proper order by the subscripted array assignment y = x(iperm(1:n)).

ICYCLE = icycle (Output)
Permutations applied to the input data are converted to cyclic interchanges. Thus, the output array y is given by the following elementary interchanges, where :=: denotes a swap:

```
j = icycle(i)
y(j) :=: y(i), i = 1,n
```

IOPT = iopt(:) (Input)
Derived type array with the same precision as the input matrix; used for passing optional data to the routine. The options are as follows:

Packaged Options for SORT_REAL		
Option Prefix = ?	Option Name	Option Value
s_, d_	Sort_real_scan_for_NaN	1

`iopt(IO) = ?_options(?_sort_real_scan_for_NaN, ?_dummy)`

Examines each input array entry to find the first value such that

`isNaN(x(i)) == .true.`
See the `isNaN()` function, Chapter 10.
Default: Does not scan for NaNs.

FORTRAN 90 Interface

Generic: `CALL SORT_REAL (X, Y [,…])`

Specific: The specific interface names are `S_SORT_REAL` and `D_SORT_REAL`.

Example 1: Sorting an Array

An array of random numbers is obtained. The values are sorted so they are nondecreasing.

```
      use sort_real_int
      use rand_gen_int

      implicit none

! This is Example 1 for SORT_REAL.

      integer, parameter :: n=100
      real(kind(1e0)), dimension(n) :: x, y

! Generate random data to sort.
      call rand_gen(x)

! Sort the data so it is non-decreasing.
      call sort_real(x, y)

! Check that the sorted array is not decreasing.
      if (count(y(1:n-1) > y(2:n)) == 0) then
         write (*,*) 'Example 1 for SORT_REAL is correct.'
      end if

      end
```

Output

```
Example 1 for SORT_REAL is correct.
```

Description

For a detailed description, see the “Description” section of routine SVRGN on page 1607, which appears later in this chapter.

Additional Examples

Example 2: Sort and Final Move with a Permutation

A set of *n* random numbers is sorted so the results are nonincreasing. The columns of an *n* × *n* random matrix are moved to the order given by the permutation defined by the interchange of the entries. Since the routine sorts the results to be algebraically nondecreasing, the array of negative values is used as input. Thus, the negative value of the sorted output order is nonincreasing. The optional argument “iperm=” records the final order and is used to move the matrix columns to that order. This example illustrates the principle of sorting record *keys*, followed by direct movement of the records to sorted order.

```
      use sort_real_int
      use rand_gen_int

      implicit none

! This is Example 2 for SORT_REAL.

      integer i
      integer, parameter :: n=100
      integer ip(n)
      real(kind(1e0)) a(n,n), x(n), y(n), temp(n*n)

! Generate a random array and matrix of values.
      call rand_gen(x)
      call rand_gen(temp)
      a = reshape(temp,(/n,n/))

! Initialize permutation to the identity.
      do i=1, n
         ip(i) = i
      end do

! Sort using negative values so the final order is
! non-increasing.
      call sort_real(-x, y, iperm=ip)

! Final movement of keys and matrix columns.
      y = x(ip(1:n))
      a = a(:,ip(1:n))

! Check the results.
      if (count(y(1:n-1) < y(2:n)) == 0) then
         write (*,*) 'Example 2 for SORT_REAL is correct.'
      end if

      end
```

Output

```
Example 2 for SORT_REAL is correct.
```

Fatal and Terminal Error Messages

See the *messages.gls* file for error messages for `sort_real`. These error messages are numbered 561−567; 581−587.

SVRGN

Sorts a real array by algebraically increasing value.

Required Arguments

RA — Vector of length `N` containing the array to be sorted. (Input)

RB — Vector of length `N` containing the sorted array. (Output)
If `RA` is not needed, `RA` and `RB` can share the same storage locations.

Optional Arguments

N — Number of elements in the array to be sorted. (Input)
Default: `N` = size (`RA`,1).

FORTRAN 90 Interface

Generic: `CALL SVRGN (RA, RB [,…])`

Specific: The specific interface names are `S_SVRGN` and `D_SVRGN`.

FORTRAN 77 Interface

Single: `CALL SVRGN (N, RA, RB)`

Double: The double precision name is `DSVRGN`.

Example

This example sorts the 10-element array `RA` algebraically.

```
      USE SVRGN_INT
      USE UMACH_INT
!                                 Declare variables
      PARAMETER  (N=10)
      REAL       RA(N), RB(N)
!                                 Set values for  RA
!     RA = ( -1.0  2.0  -3.0  4.0  -5.0  6.0  -7.0  8.0  -9.0  10.0 )
!
      DATA RA/-1.0, 2.0, -3.0, 4.0, -5.0, 6.0, -7.0, 8.0, -9.0, 10.0/
```

```
!                                  Sort RA by algebraic value into RB
      CALL SVRGN (RA, RB)
!                                  Print results
      CALL UMACH (2,NOUT)
      WRITE (NOUT, 99999) (RB(J),J=1,N)
!
99999 FORMAT (’  The output vector is:’, /, 10(1X,F5.1))
      END
```

Output

```
The Output vector is:
-9.0  -7.0  -5.0  -3.0  -1.0   2.0   4.0   6.0   8.0  10.0
```

Description

Routine SVRGN sorts the elements of an array, *A*, into ascending order by algebraic value. The array *A* is divided into two parts by picking a central element *T* of the array. The first and last elements of *A* are compared with *T* and exchanged until the three values appear in the array in ascending order. The elements of the array are rearranged until all elements greater than or equal to the central element appear in the second part of the array and all those less than or equal to the central element appear in the first part. The upper and lower subscripts of one of the segments are saved, and the process continues iteratively on the other segment. When one segment is finally sorted, the process begins again by retrieving the subscripts of another unsorted portion of the array. On completion, $A_j \le A_i$ for $j < i$. For more details, see Singleton (1969), Griffin and Redish (1970), and Petro (1970).

SVRGP

Sorts a real array by algebraically increasing value and return the permutation that rearranges the array.

Required Arguments

RA — Vector of length N containing the array to be sorted. (Input)

RB — Vector of length N containing the sorted array. (Output)
If RA is not needed, RA and RB can share the same storage locations.

IPERM — Vector of length N. (Input/Output)
On input, IPERM should be initialized to the values 1, 2, …, N. On output, IPERM contains a record of permutations made on the vector RA.

Optional Arguments

N — Number of elements in the array to be sorted. (Input)
Default: N = size (IPERM,1).

FORTRAN 90 Interface

Generic: CALL SVRGP (RA, RB, IPERM [,…])

Specific: The specific interface names are S_SVRGP and D_SVRGP.

FORTRAN 77 Interface

Single: CALL SVRGP (N, RA, RB, IPERM)

Double: The double precision name is DSVRGP.

Example

This example sorts the 10-element array RA algebraically.

```
      USE SVRGP_INT
      USE UMACH_INT
!                                 Declare variables
      PARAMETER  (N=10)
      REAL       RA(N), RB(N)
      INTEGER    IPERM(N)
!                                 Set values for  RA and IPERM
!     RA    = ( 10.0  -9.0  8.0  -7.0  6.0  5.0  4.0  -3.0  -2.0  -1.0 )
!
!     IPERM = ( 1  2  3  4  5  6  7  8  9  10)
!
      DATA RA/10.0, -9.0, 8.0, -7.0, 6.0, 5.0, 4.0, -3.0, -2.0, -1.0/
      DATA IPERM/1, 2, 3, 4, 5, 6, 7, 8, 9, 10/
!                                 Sort RA by algebraic value into RB
      CALL SVRGP (RA, RB, IPERM)
!                                 Print results
      CALL UMACH (2,NOUT)
      WRITE (NOUT, 99998) (RB(J),J=1,N)
      WRITE (NOUT, 99999) (IPERM(J),J=1,N)
!
99998 FORMAT ('  The output vector is:', /, 10(1X,F5.1))
99999 FORMAT ('  The permutation vector is:', /, 10(1X,I5))
      END
```

Output

```
The output vector is:
-9.0  -7.0  -3.0  -2.0  -1.0   4.0   5.0   6.0   8.0  10.0

The permutation vector is:
2     4     8     9    10     7     6     5     3     1
```

Comments

For wider applicability, integers (1, 2, …, N) that are to be associated with RA(I) for I = 1, 2, …, N may be entered into IPERM(I) in any order. Note that these integers must be unique.

Description

Routine SVRGP sorts the elements of an array, *A*, into ascending order by algebraic value, keeping a record in *P* of the permutations to the array *A*. That is, the elements of *P* are moved in the same manner as are the elements in *A* as *A* is being sorted. The routine SVRGP uses the algorithm discussed in SVRGN (page 1604). On completion, $A_j \le A_i$ for $j < i$.

SVIGN

Sorts an integer array by algebraically increasing value.

Required Arguments

IA — Integer vector of length N containing the array to be sorted.   (Input)

IB — Integer vector of length N containing the sorted array.   (Output)
If IA is not needed, IA and IB can share the same storage locations.

Optional Arguments

N — Number of elements in the array to be sorted.   (Input)
Default: N = size (IA,1).

FORTRAN 90 Interface

Generic: CALL SVIGN (IA, IB [,…])

Specific: The specific interface name is S_SVIGN .

FORTRAN 77 Interface

Single: CALL SVIGN (N, IA, IB)

Example

This example sorts the 10-element array IA algebraically.

```
      USE SVIGN_INT
      USE UMACH_INT
!                                 Declare variables
      PARAMETER  (N=10)
      INTEGER    IA(N), IB(N)
!                                 Set values for  IA
!     IA = ( -1  2  -3  4  -5  6  -7  8  -9  10 )
!
      DATA IA/-1, 2, -3, 4, -5, 6, -7, 8, -9, 10/
!                                 Sort IA by algebraic value into IB
      CALL SVIGN (IA, IB)
!                                 Print results
```

```
      CALL UMACH (2,NOUT)
      WRITE (NOUT, 99999) (IB(J),J=1,N)
!
99999 FORMAT ('  The output vector is:', /, 10(1X,I5))
      END
```

Output

```
The Output vector is:
-9    -7    -5    -3    -1     2     4     6     8    10
```

Description

Routine SVIGN sorts the elements of an integer array, *A*, into ascending order by algebraic value. The routine SVIGN uses the algorithm discussed in SVRGN (page 1604). On completion, $A_j \le A_i$ for $j < i$.

SVIGP

Sorts an integer array by algebraically increasing value and return the permutation that rearranges the array.

Required Arguments

IA — Integer vector of length N containing the array to be sorted. (Input)

IB — Integer vector of length N containing the sorted array. (Output)
If IA is not needed, IA and IB can share the same storage locations.

IPERM — Vector of length N. (Input/Output)
On input, IPERM should be initialized to the values 1, 2, …, N. On output, IPERM contains a record of permutations made on the vector IA.

Optional Arguments

N — Number of elements in the array to be sorted. (Input)
Default: N = size (IPERM,1).

FORTRAN 90 Interface

Generic: CALL SVIGP (IA, IB, IPERM [,…])

Specific: The specific interface name is S_SVIGP.

FORTRAN 77 Interface

Single: CALL SVIGP (N, IA, IB, IPERM)

Example

This example sorts the 10-element array IA algebraically.

```
      USE SVIGP_INT
      USE UMACH_INT
!                                 Declare variables
      PARAMETER  (N=10)
      INTEGER    IA(N), IB(N), IPERM(N)
!                                 Set values for  IA and IPERM
!     IA    = ( 10  -9  8  -7  6  5  4  -3  -2  -1 )
!
!     IPERM = ( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 )
!
      DATA IA/10, -9, 8, -7, 6, 5, 4, -3, -2, -1/
      DATA IPERM/1, 2, 3, 4, 5, 6, 7, 8, 9, 10/
!                                 Sort IA by algebraic value into IB
      CALL SVIGP (IA, IB, IPERM)
!                                 Print results
      CALL UMACH (2,NOUT)
      WRITE (NOUT, 99998) (IB(J),J=1,N)
      WRITE (NOUT, 99999) (IPERM(J),J=1,N)
!
99998 FORMAT (' The output vector is:', /, 10(1X,I5))
99999 FORMAT (' The permutation vector is:', /, 10(1X,I5))
      END
```

Output

```
The Output vector is:
-9    -7    -3    -2    -1     4     5     6     8    10

The permutation vector is:
2     4     8     9    10     7     6     5     3     1
```

Comments

For wider applicability, integers (1, 2, …, N) that are to be associated with IA(I) for I = 1, 2, …, N may be entered into IPERM(I) in any order. Note that these integers must be unique.

Description

Routine SVIGP sorts the elements of an integer array, *A*, into ascending order by algebraic value, keeping a record in *P* of the permutations to the array *A*. That is, the elements of *P* are moved in the same manner as are the elements in *A* as *A* is being sorted. The routine SVIGP uses the algorithm discussed in SVRGN (page 1604). On completion, $A_j \le A_i$ for $j < i$.

SVRBN

Sorts a real array by nondecreasing absolute value.

Required Arguments

RA — Vector of length N containing the array to be sorted. (Input)

RB — Vector of length N containing the sorted array. (Output)
If RA is not needed, RA and RB can share the same storage locations.

Optional Arguments

N — Number of elements in the array to be sorted. (Input)
Default: N = size (RA,1).

FORTRAN 90 Interface

Generic: CALL SVRBN (RA, RB [,…])

Specific: The specific interface names are S_SVRBN and D_SVRBN.

FORTRAN 77 Interface

Single: CALL SVRBN (N, RA, RB)

Double: The double precision name is DSVRBN.

Example

This example sorts the 10-element array RA by absolute value.

```
      USE SVRBN_INT
      USE UMACH_INT
!                                 Declare variables
      PARAMETER  (N=10)
      REAL       RA(N), RB(N)
!                                 Set values for  RA
!       RA = ( -1.0  3.0  -4.0  2.0  -1.0  0.0  -7.0  6.0  10.0  -7.0 )
!
      DATA RA/-1.0, 3.0, -4.0, 2.0, -1.0, 0.0, -7.0, 6.0, 10.0, -7.0/
!                                 Sort RA by absolute value into RB
      CALL SVRBN (RA, RB)
!                                 Print results
      CALL UMACH (2,NOUT)
      WRITE (NOUT, 99999) (RB(J),J=1,N)
!
99999 FORMAT ('  The output vector is :', /, 10(1X,F5.1))
      END
```

Output

```
The Output vector is :
0.0  -1.0  -1.0   2.0   3.0  -4.0   6.0  -7.0  -7.0  10.0
```

Description

Routine SVRBN sorts the elements of an array, A, into ascending order by absolute value. The routine SVRBN uses the algorithm discussed in SVRGN (page 1604). On completion, $|A_j| \le |A_i|$ for $j < i$.

SVRBP

Sorts a real array by nondecreasing absolute value and return the permutation that rearranges the array.

Required Arguments

RA — Vector of length N containing the array to be sorted. (Input)

RB — Vector of length N containing the sorted array. (Output)
If RA is not needed, RA and RB can share the same storage locations.

IPERM — Vector of length N. (Input/Output)
On input, IPERM should be initialized to the values 1, 2, …, N. On output, IPERM contains a record of permutations made on the vector IA.

Optional Arguments

N — Number of elements in the array to be sorted. (Input)
Default: N = size (IPERM,1).

FORTRAN 90 Interface

Generic: CALL SVRBP (RA, RB, IPERM[,…])

Specific: The specific interface names are S_SVRBP and D_SVRBP.

FORTRAN 77 Interface

Single: CALL SVRBP (N, RA, RB, IPERM)

Double: The double precision name is DSVRBP.

Example

This example sorts the 10-element array RA by absolute value.

```
      USE SVRBP_INT
      USE UMACH_INT
!                                 Declare variables
      PARAMETER   (N=10)
      REAL        RA(N), RB(N)
      INTEGER    IPERM(N)
!                                Set values for  RA and IPERM
!     RA     = ( 10.0  9.0  8.0  7.0  6.0  5.0  -4.0  3.0  -2.0  1.0 )
!
!     IPERM = ( 1  2  3  4  5  6  7  8  9  10 )
!
      DATA RA/10.0, 9.0, 8.0, 7.0, 6.0, 5.0, -4.0, 3.0, -2.0, 1.0/
      DATA IPERM/1, 2, 3, 4, 5, 6, 7, 8, 9, 10/
!                                Sort RA by absolute value into RB
      CALL SVRBP (RA, RB, IPERM)
!                                Print results
      CALL UMACH (2,NOUT)
      WRITE (NOUT, 99998) (RB(J),J=1,N)
      WRITE (NOUT, 99999) (IPERM(I),I=1,N)
!
99998 FORMAT ('  The output vector is:', /, 10(1X,F5.1))
99999 FORMAT ('  The permutation vector is:', /, 10(1X,I5))
      END
```

Output

```
The output vector is:
1.0  -2.0   3.0  -4.0   5.0   6.0   7.0   8.0   9.0  10.0
The permutation vector is:
10      9     8     7     6     5     4     3     2     1
```

Comments

For wider applicability, integers (1, 2, …, N) that are to be associated with RA(I) for I = 1, 2, …, N may be entered into IPERM(I) in any order. Note that these integers must be unique.

Description

Routine SVRBP sorts the elements of an array, *A*, into ascending order by absolute value, keeping a record in *P* of the permutations to the array *A*. That is, the elements of *P* are moved in the same manner as are the elements in *A* as *A* is being sorted. The routine SVRBP uses the algorithm discussed in SVRGN (page 1604). On completion, $A_j \le A_i$ for $j < i$.

SVIBN

Sorts an integer array by nondecreasing absolute value.

Required Arguments

IA — Integer vector of length N containing the array to be sorted. (Input)

IB — Integer vector of length N containing the sorted array. (Output)
If IA is not needed, IA and IB can share the same storage locations.

Optional Arguments

N — Number of elements in the array to be sorted. (Input)
Default: N = size (IA,1).

FORTRAN 90 Interface

Generic: CALL SVIBN (IA, IB [,…])

Specific: The specific interface name is S_SVIBN.

FORTRAN 77 Interface

Single: CALL SVIBN (N, IA, IB)

Example

This example sorts the 10-element array IA by absolute value.

```
      USE SVIBN_INT
      USE UMACH_INT
!                                 Declare variables
      PARAMETER  (N=10)
      INTEGER    IA(N), IB(N)
!                                 Set values for  IA
!     IA = ( -1  3  -4  2  -1  0  -7  6  10  -7)
!
      DATA IA/-1, 3, -4, 2, -1, 0, -7, 6, 10, -7/
!                                 Sort IA by absolute value into IB
      CALL SVIBN (IA, IB)
!                                 Print results
      CALL UMACH (2,NOUT)
      WRITE (NOUT, 99999) (IB(J),J=1,N)
!
99999 FORMAT ('  The output vector is:', /, 10(1X,I5))
      END
```

Output

```
The Output vector is:
0    -1    -1     2     3    -4     6    -7    -7    10
```

Description

Routine SVIBN sorts the elements of an integer array, *A*, into ascending order by absolute value. This routine SVIBN uses the algorithm discussed in SVRGN (page 1604). On completion, $A_j \le A_i$ for $j < i$.

SVIBP

Sorts an integer array by nondecreasing absolute value and return the permutation that rearranges the array.

Required Arguments

IA — Integer vector of length `N` containing the array to be sorted. (Input)

IB — Integer vector of length `N` containing the sorted array. (Output)
If `IA` is not needed, `IA` and `IB` can share the same storage locations.

IPERM — Vector of length `N`. (Input/Output)
On input, `IPERM` should be initialized to the values 1, 2, …, `N`. On output, `IPERM` contains a record of permutations made on the vector `IA`.

Optional Arguments

N — Number of elements in the array to be sorted. (Input)
Default: `N` = size (`IA`,1).

FORTRAN 90 Interface

Generic: `CALL SVIBP (IA, IB, IPERM [,…])`

Specific: The specific interface name is `S_SVIBP`.

FORTRAN 77 Interface

Single: `CALL SVIBP (N, IA, IB, IPERM)`

Example

This example sorts the 10-element array `IA` by absolute value.

```
      USE SVIBP_INT
      USE UMACH_INT
!                                 Declare variables
      PARAMETER  (N=10)
      INTEGER    IA(N), IB(N), IPERM(N)
!                                 Set values for  IA
!     IA    = ( 10  9  8  7  6  5  -4  3  -2  1 )
!
!     IPERM = ( 1  2  3  4  5  6  7  8  9  10 )
!
      DATA IA/10, 9, 8, 7, 6, 5, -4, 3, -2, 1/
      DATA IPERM/1, 2, 3, 4, 5, 6, 7, 8, 9, 10/
!                                 Sort IA by absolute value into IB
      CALL SVIBP (IA, IB, IPERM)
```

```
!                                  Print results
      CALL UMACH (2,NOUT)
      WRITE (NOUT, 99998) (IB(J),J=1,N)
      WRITE (NOUT, 99999) (IPERM(J),J=1,N)
!
99998 FORMAT ('  The output vector is:', /, 10(1X,I5))
99999 FORMAT ('  The permutation vector is:', /, 10(1X,I5))
      END
```

Output

```
The Output vector is:
1    -2     3    -4     5     6     7     8     9    10

The permutation vector is:
10     9     8     7     6     5     4     3     2     1
```

Comments

For wider applicability, integers (1, 2, …, N) that are to be associated with IA(I) for I = 1, 2, …, N may be entered into IPERM(I) in any order. Note that these integers must be unique.

Description

Routine SVIBP sorts the elements of an integer array, A, into ascending order by absolute value, keeping a record in P of the permutations to the array A. That is, the elements of P are moved in the same manner as are the elements in A as A is being sorted. The routine SVIBP uses the algorithm discussed in SVRGN (page 1604). On completion, $A_j \le A_i$ for $j < i$.

SRCH

Searches a sorted vector for a given scalar and return its index.

Required Arguments

VALUE — Scalar to be searched for in Y.   (Input)

X — Vector of length N * INCX.   (Input)
Y is obtained from X for I = 1, 2, …, N by Y(I) = X(1 + (I − 1) * INCX). Y(1), Y(2), …, Y(N) must be in ascending order.

INDEX — Index of Y pointing to VALUE.   (Output)
If INDEX is positive, VALUE is found in Y. If INDEX is negative, VALUE is not found in Y.

INDEX	**Location of VALUE**
1 thru N	VALUE = Y(INDEX)

−1	VALUE < Y(1) or N = 0
−N thru −2	Y(−INDEX − 1) < VALUE < Y(INDEX)
−(N + 1)	VALUE > Y(N)

Optional Arguments

N — Length of vector Y. (Input)
Default: N = (size (X,1)) / INCX.

INCX — Displacement between elements of X. (Input)
INCX must be greater than zero.
Default: INCX = 1.

FORTRAN 90 Interface

Generic: CALL SRCH (VALUE, X, INDEX [,…])

Specific: The specific interface names are S_SRCH and D_SRCH.

FORTRAN 77 Interface

Single: CALL SRCH (N, VALUE, X, INCX, INDEX)

Double: The double precision name is DSRCH.

Example

This example searches a real vector sorted in ascending order for the value 653.0. The problem is discussed by Knuth (1973, pages 407–409).

```
      USE SRCH_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=16)
!
      INTEGER    INDEX, NOUT
      REAL       VALUE, X(N)
!
      DATA X/61.0, 87.0, 154.0, 170.0, 275.0, 426.0, 503.0, 509.0, &
          512.0, 612.0, 653.0, 677.0, 703.0, 765.0, 897.0, 908.0/
!
      VALUE = 653.0
      CALL SRCH (VALUE, X, INDEX)
!
      CALL UMACH (2, NOUT)
      WRITE (NOUT,*) ’INDEX = ’, INDEX
      END
```

Output

```
INDEX =   11
```

Description

Routine `SRCH` searches a real vector x (stored in `X`), whose n elements are sorted in ascending order for a real number c (stored in `VALUE`). If c is found in x, its index i (stored in `INDEX`) is returned so that $x_i = c$. Otherwise, a negative number i is returned for the index. Specifically,

if $1 \le i \le n$	then $x_i = c$
if $i = -1$	then $c < x_1$ or $n = 0$
if $-n \le I \le -2$	then $x_{-i-1} < c < x_{-i}$
if $i = -(n + 1)$	then $c > x_n$

The argument `INCX` is useful if a row of a matrix, for example, row number `I` of a matrix `X`, must be searched. The elements of row `I` are assumed to be in ascending order. In this case, set `INCX` equal to the leading dimension of `X` exactly as specified in the dimension statement in the calling program. With `X` declared

```
REAL X(LDX,N)
```

the invocation

```
CALL SRCH (N, VALUE, X(I,1), LDX, INDEX)
```

returns an index that will reference a column number of `X`.

Routine `SRCH` performs a binary search. The routine is an implementation of algorithm *B* discussed by Knuth (1973, pages 407–411).

ISRCH

Searches a sorted integer vector for a given integer and return its index.

Required Arguments

IVALUE — Scalar to be searched for in `IY`. (Input)

IX — Vector of length `N * INCX`. (Input)
`IY` is obtained from `IX` for `I` = 1, 2, …, `N` by `IY(I)` = `IX`(1 + (`I` – 1) `* INCX`). `IY`(1), `IY`(2), …, `IY`(`N`) must be in ascending order.

INDEX — Index of `IY` pointing to `IVALUE`. (Output)
If `INDEX` is positive, `IVALUE` is found in `IY`. If `INDEX` is negative, `IVALUE` is not found in `IY`.

`INDEX`	**Location of `VALUE`**

1 thru N	IVALUE = IY(INDEX)
−1	IVALUE < IY(1) or N = 0
−N thru −2	IY(−INDEX − 1) < IVALUE < IY(−INDEX)
−(N + 1)	IVALUE > Y(N)

Optional Arguments

N — Length of vector IY. (Input)
Default: N = size (IX,1) / INCX.

INCX — Displacement between elements of IX. (Input)
INCX must be greater than zero.
Default: INCX = 1.

FORTRAN 90 Interface

Generic: CALL ISRCH (IVALUE, IX, INDEX [,…])

Specific: The specific interface name is S_ISRCH.

FORTRAN 77 Interface

Single: CALL ISRCH (N, IVALUE, IX, INCX, INDEX)

Example

This example searches an integer vector sorted in ascending order for the value 653. The problem is discussed by Knuth (1973, pages 407–409).

```
      USE ISRCH_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=16)
!
      INTEGER    INDEX, NOUT
      INTEGER    IVALUE, IX(N)
!
      DATA IX/61, 87, 154, 170, 275, 426, 503, 509, 512, 612, 653, 677, &
             703, 765, 897, 908/
!
      IVALUE = 653
      CALL ISRCH (IVALUE, IX, INDEX)
!
      CALL UMACH (2, NOUT)
      WRITE (NOUT,*) 'INDEX = ', INDEX
      END
```

Output

```
INDEX =   11
```

Description

Routine ISRCH searches an integer vector x (stored in IX), whose n elements are sorted in ascending order for an integer c (stored in IVALUE). If c is found in x, its index i (stored in INDEX) is returned so that $x_i = c$. Otherwise, a negative number i is returned for the index. Specifically,

if $1 \le i \le n$	Then $x_i = c$
if $i = -1$	Then $c < x_1$ or $n = 0$
if $-n \le i \le -2$	Then $x_{-i-1} < c < x_{-i}$
if $i = -(n + 1)$	Then $c > x_n$

The argument INCX is useful if a row of a matrix, for example, row number I of a matrix IX, must be searched. The elements of row I are assumed to be in ascending order. Here, set INCX equal to the leading dimension of IX exactly as specified in the dimension statement in the calling program. With IX declared

```
INTEGER IX(LDIX,N)
```

the invocation

```
CALL ISRCH (N, IVALUE, IX(I,1), LDIX, INDEX)
```

returns an index that will reference a column number of IX.

The routine ISRCH performs a binary search. The routine is an implementation of algorithm *B* discussed by Knuth (1973, pages 407–411).

SSRCH

Searches a character vector, sorted in ascending ASCII order, for a given string and return its index.

Required Arguments

N — Length of vector CHY. (Input)
Default: N = size (CHX,1) / INCX.

STRING — Character string to be searched for in CHY. (Input)

CHX — Vector of length N * INCX containing character strings. (Input)
CHY is obtained from CHX for I = 1, 2, …, N by CHY(I) = CHX(1 + (I – 1) * INCX).
CHY(1), CHY(2), …, CHY(N) must be in ascending ASCII order.

INCX — Displacement between elements of CHX. (Input)
INCX must be greater than zero.
Default: INCX = 1.

INDEX — Index of CHY pointing to STRING. (Output)
If INDEX is positive, STRING is found in CHY. If INDEX is negative, STRING is not found in CHY.

INDEX	**Location of STRING**
1 thru N	STRING = CHY(INDEX)
−1	STRING < CHY(1) or N = 0
−N thru −2	CHY(−INDEX − 1) < STRING < CHY(−INDEX)
−(N + 1)	STRING > CHY(N)

FORTRAN 90 Interface

Generic: CALL SSRCH (N, STRING, CHX, INCX, INDEX)

Specific: The specific interface name is SSRCH.

FORTRAN 77 Interface

Single: CALL SSRCH (N, STRING, CHX, INCX, INDEX)

Example

This example searches a CHARACTER * 2 vector containing 9 character strings, sorted in ascending ASCII order, for the value ′CC′.

```
      USE SSRCH_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=9)

!
      INTEGER    INDEX, NOUT
      CHARACTER  CHX(N)*2, STRING*2
!
      DATA CHX/′AA′, ′BB′, ′CC′, ′DD′, ′EE′, ′FF′, ′GG′, ′HH′, &
          ′II′/
!
      INCX   = 1
      STRING = ′CC′
      CALL SSRCH (N, STRING, CHX, INCX, INDEX)
!
      CALL UMACH (2, NOUT)
      WRITE (NOUT,*) ′INDEX = ′, INDEX
      END
```

Output

```
INDEX =   3
```

Description

Routine SSRCH searches a vector of character strings *x* (stored in CHX), whose *n* elements are sorted in ascending ASCII order, for a character string *c* (stored in STRING). If *c* is found in *x*, its index *i* (stored in INDEX) is returned so that $x_i = c$. Otherwise, a negative number *i* is returned for the index. Specifically,

if $1 \le i \le n$	Then $x_i = c$
if $i = -1$	Then $c < x_1$ or $n = 0$
if $-n \le I \le -2$	Then $x_{-i-1} < c < x_{-i}$
if $i = -(n + 1)$	Then $c > x_n$

Here, “<“ and “>” are in reference to the ASCII collating sequence. For comparisons made between character strings *c* and x_i with different lengths, the shorter string is considered as if it were extended on the right with blanks to the length of the longer string. (SSRCH uses FORTRAN intrinsic functions LLT and LGT.)

The argument INCX is useful if a row of a matrix, for example, row number I of a matrix CHX, must be searched. The elements of row I are assumed to be in ascending ASCII order. In this case, set INCX equal to the leading dimension of CHX exactly as specified in the dimension statement in the calling program. With CHX declared

```
CHARACTER * 7 CHX(LDCHX,N)
```

the invocation

```
CALL SSRCH (N, STRING, CHX(I,1), LDCHX, INDEX)
```

returns an index that will reference a column number of CHX.

Routine SSRCH performs a binary search. The routine is an implementation of algorithm *B* discussed by Knuth (1973, pages 407–411).

ACHAR

This function returns a character given its ASCII value.

Function Return Value

ACHAR — CHARACTER * 1 string containing the character in the I-th position of the ASCII collating sequence. (Output)

Required Arguments

I — Integer ASCII value of the character desired. (Input)
I must be greater than or equal to zero and less than or equal to 127.

FORTRAN 90 Interface

Generic: `ACHAR (I)`

Specific: The specific interface name is `ACHAR`.

FORTRAN 77 Interface

Single: `ACHAR (I)`

Example

This example returns the character of the ASCII value 65.

```
      USE ACHAR_INT
      USE UMACH_INT
      INTEGER    I, NOUT
!
      CALL UMACH (2, NOUT)
!                                 Get character for ASCII value
!                                 of 65 ('A')
      I = 65
      WRITE (NOUT,99999) I, ACHAR(I)
!
99999 FORMAT (' For the ASCII value of ', I2, ', the character is : ', &
            A1)
      END
```

Output

```
For the ASCII value of 65, the character is : A
```

Description

Routine `ACHAR` returns the character of the input ASCII value. The input value should be between 0 and 127. If the input value is out of range, the value returned in `ACHAR` is machine dependent.

IACHAR

This function returns the integer ASCII value of a character argument.

Function Return Value

IACHAR — Integer ASCII value for `CH`. (Output)
The character `CH` is in the `IACHAR`-th position of the ASCII collating sequence.

Required Arguments

CH — Character argument for which the integer ASCII value is desired. (Input)

FORTRAN 90 Interface

Generic: `IACHAR(CH)`

Specific: The specific interface name is `IACHAR`.

FORTRAN 77 Interface

Single: `IACHAR(CH)`

Example

This example gives the ASCII value of character A.

```
      USE IACHAR_INT
      INTEGER    NOUT
      CHARACTER  CH
!
      CALL UMACH (2, NOUT)
!                                 Get ASCII value for the character
!                                 ’A’.
      CH = ’A’
      WRITE (NOUT,99999) CH, IACHAR(CH)
!
99999 FORMAT (’ For the character  ’, A1, ’  the ASCII value is : ’, &
            I3)
      END
```

Output

```
For the character  A  the ASCII value is :  65
```

Description

Routine `IACHAR` returns the ASCII value of the input character.

ICASE

This function returns the ASCII value of a character converted to uppercase.

Function Return Value

ICASE — Integer ASCII value for `CH` without regard to the case of `CH`. (Output)
Routine `ICASE` returns the same value as `IACHAR` (page 1625) for all but lowercase letters. For these, it returns the `IACHAR` value for the corresponding uppercase letter.

Required Arguments

CH — Character to be converted. (Input)

FORTRAN 90 Interface

Generic: `ICASE(CH)`

Specific: The specific interface name is `ICASE`.

FORTRAN 77 Interface

Single: `ICASE(CH)`

Example

This example shows the case insensitive conversion.

```
      USE ICASE_INT
      USE UMACH_INT
      INTEGER    NOUT
      CHARACTER  CHR
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Get ASCII value for the character
!                                 ’a’.
      CHR = ’a’
      WRITE (NOUT,99999) CHR, ICASE(CHR)
!
99999 FORMAT (’ For the character  ’, A1, ’  the ICASE value is : ’, &
            I3)
      END
```

Output

```
For the character  a  the ICASE value is :  65
```

Description

Routine `ICASE` converts a character to its integer ASCII value. The conversion is case insensitive; that is, it returns the ASCII value of the corresponding uppercase letter for a lowercase letter.

IICSR

This function compares two character strings using the ASCII collating sequence but without regard to case.

Function Return Value

IICSR — Comparison indicator. (Output)
Let `USTR1` and `USTR2` be the uppercase versions of `STR1` and `STR2`, respectively. The following table indicates the relationship between `USTR1` and `USTR2` as determined by the ASCII collating sequence.

IICSR	**Meaning**
–1	USTR1 precedes USTR2
0	USTR1 equals USTR2
1	USTR1 follows USTR2

Required Arguments

STR1 — First character string. (Input)

STR2 — Second character string. (Input)

FORTRAN 90 Interface

Generic: `IICSR(STR1, STR2)`

Specific: The specific interface name is `IICSR`.

FORTRAN 77 Interface

Single: `IICSR(STR1, STR2)`

Example

This example shows different cases on comparing two strings.

```
      USE IICSR_INT
      USE UMACH_INT
      INTEGER    NOUT
      CHARACTER  STR1*6, STR2*6
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Compare String1 and String2
!                                 String1 is ’bigger’ than String2
      STR1 = ’ABc 1’
      STR2 = ’ ’
      WRITE (NOUT,99999) STR1, STR2, IICSR(STR1,STR2)
!
!                                 String1 is ’equal’ to String2
      STR1 = ’AbC’
      STR2 = ’ABc’
      WRITE (NOUT,99999) STR1, STR2, IICSR(STR1,STR2)
!
!                                 String1 is ’smaller’ than String2
      STR1 = ’ABc’
      STR2 = ’aBC 1’
      WRITE (NOUT,99999) STR1, STR2, IICSR(STR1,STR2)
!
99999 FORMAT (’ For String1 = ’, A6, ’and String2 = ’, A6, &
     ' IICSR = ', I2, /)
END
```

Output

```
For String1 = ABc 1 and String2 =        IICSR =  1

For String1 = AbC   and String2 = ABc    IICSR =  0

For String1 = ABc   and String2 = aBC 1  IICSR = -1
```

Comments

If the two strings, STR1 and STR2, are of unequal length, the shorter string is considered as if it were extended with blanks to the length of the longer string.

Description

Routine IICSR compares two character strings. It returns –1 if the first string is less than the second string, 0 if they are equal, and 1 if the first string is greater than the second string. The comparison is case insensitive.

IIDEX

This funcion determines the position in a string at which a given character sequence begins without regard to case.

Function Return Value

IIDEX — Position in CHRSTR where KEY begins. (Output)
If KEY occurs more than once in CHRSTR, the starting position of the first occurrence is returned. If KEY does not occur in CHRSTR, then IIDEX returns a zero.

Required Arguments

CHRSTR — Character string to be searched. (Input)

KEY — Character string that contains the key sequence. (Input)

FORTRAN 90 Interface

Generic: IIDEX(CHRSTR, KEY)

Specific: The specific interface name is IIDEX.

FORTRAN 77 Interface

Single: IIDEX(CHRSTR, KEY)

Example

This example locates a key string.

```
      USE IIDEX_INT
      USE UMACH_INT
      INTEGER    NOUT
      CHARACTER  KEY*5, STRING*10
!                                  Get output unit number
      CALL UMACH (2, NOUT)
!                                  Locate KEY in STRING
      STRING = ’a1b2c3d4e5’
      KEY    = ’C3d4E’
      WRITE (NOUT,99999) STRING, KEY, IIDEX(STRING,KEY)
!
      KEY = ’F’
      WRITE (NOUT,99999) STRING, KEY, IIDEX(STRING,KEY)
!
99999 FORMAT (’ For STRING = ’, A10, ’ and KEY = ’, A5, ’ IIDEX = ’, I2, &
            /)
      END
```

Output

```
For STRING = a1b2c3d4e5 and KEY = C3d4E IIDEX =  5

For STRING = a1b2c3d4e5 and KEY = F     IIDEX =  0
```

Comments

If the length of KEY is greater than the length CHRSTR, IIDEX returns a zero.

Description

Routine IIDEX searches for a key string in a given string and returns the index of the starting element at which the key character string begins. It returns 0 if there is no match. The comparison is case insensitive. For a case-sensitive version, use the FORTRAN 77 intrinsic function INDEX.

CVTSI

Converts a character string containing an integer number into the corresponding integer form.

Required Arguments

STRING — Character string containing an integer number. (Input)

NUMBER — The integer equivalent of STRING. (Output)

FORTRAN 90 Interface

Generic: CALL CVTSI (STRING, NUMBER)

Specific: The specific interface name is CVTSI.

FORTRAN 77 Interface

Single: CALL CVTSI (STRING, NUMBER)

Example

The string “12345” is converted to an INTEGER variable.

```
      USE CVTSI_INT
      USE UMACH_INT
      INTEGER    NOUT, NUMBER
      CHARACTER  STRING*10
!
      DATA STRING/’12345’/
!
      CALL CVTSI (STRING, NUMBER)
!
      CALL UMACH (2, NOUT)
      WRITE (NOUT,*) ’NUMBER = ’, NUMBER
      END
```

Output

```
NUMBER =   12345
```

Description

Routine CVTSI converts a character string containing an integer to an INTEGER variable. Leading and trailing blanks in the string are ignored. If the string contains something other than an integer, a terminal error is issued. If the string contains an integer larger than can be represented by an INTEGER variable as determined from routine IMACH (see the Reference Material), a terminal error is issued.

CPSEC

This fuction returns CPU time used in seconds.

Function Return Value

CPSEC — CPU time used (in seconds) since first call to CPSEC. (Output)

Required Arguments

None

FORTRAN 90 Interface

Generic: CPSEC ()

Specific: The specific interface name is CPSEC.

FORTRAN 77 Interface

Single: `CPSEC (1)`

Comments

1. The first call to CPSEC returns 0.0.

2. The accuracy of this routine depends on the hardware and the operating system. On some systems, identical runs can produce timings differing by more than 10 percent.

TIMDY

Gets time of day.

Required Arguments

IHOUR — Hour of the day. (Output)
IHOUR is between 0 and 23 inclusive.

MINUTE — Minute within the hour. (Output)
MINUTE is between 0 and 59 inclusive.

ISEC — Second within the minute. (Output)
ISEC is between 0 and 59 inclusive.

FORTRAN 90 Interface

Generic: `CALL TIMDY (IHOUR, MINUTE, ISEC)`

Specific: The specific interface name is TIMDY.

FORTRAN 77 Interface

Single: `CALL TIMDY (IHOUR, MINUTE, ISEC)`

Example

The following example uses TIMDY to return the current time. Obviously, the output is dependent upon the time at which the program is run.

```
      USE TIMDY_INT
      USE UMACH_INT
      INTEGER    IHOUR, IMIN, ISEC, NOUT
!
      CALL TIMDY (IHOUR, IMIN, ISEC)
      CALL UMACH (2, NOUT)
      WRITE (NOUT,*) 'Hour:Minute:Second = ', IHOUR, ':', IMIN, &
                   ':', ISEC
      IF (IHOUR .EQ. 0) THEN
         WRITE (NOUT,*) 'The time is ', IMIN, ' minute(s), ', ISEC, &
                      ' second(s) past midnight.'
      ELSE IF (IHOUR .LT. 12) THEN
         WRITE (NOUT,*) 'The time is ', IMIN, ' minute(s), ', ISEC, &
                      ' second(s) past ', IHOUR, ' am.'
      ELSE IF (IHOUR .EQ. 12) THEN
         WRITE (NOUT,*) 'The time is ', IMIN, ' minute(s), ', ISEC, &
                      ' second(s) past noon.'
      ELSE
         WRITE (NOUT,*) 'The time is ', IMIN, ' minute(s), ', ISEC, &
                      ' second(s) past ', IHOUR-12, ' pm.'
      END IF
      END
```

Output

```
Hour:Minute:Second =   16:  52:  29
The time is   52 minute(s),   29 second(s) past   4 pm.
```

Description

Routine TIMDY is used to retrieve the time of day.

TDATE

Gets today's date.

Required Arguments

IDAY — Day of the month. (Output)
IDAY is between 1 and 31 inclusive.

MONTH — Month of the year. (Output)
MONTH is between 1 and 12 inclusive.

IYEAR — Year. (Output)
For example, IYEAR = 1985.

FORTRAN 90 Interface

Generic: CALL TDATE (IDAY, MONTH, IYEAR)

Specific: The specific interface name is TDATE.

FORTRAN 77 Interface

Single: `CALL TDATE (IDAY, MONTH, IYEAR)`

Example

The following example uses `TDATE` to return today’s date.

```
      USE TDATE_INT
      USE UMACH_INT
      INTEGER    IDAY, IYEAR, MONTH, NOUT
!
      CALL TDATE (IDAY, MONTH, IYEAR)
      CALL UMACH (2, NOUT)
      WRITE (NOUT,*) 'Day-Month-Year = ', IDAY, '-', MONTH, &
                  '-', IYEAR
      END
```

Output

```
 Day-Month-Year =  3 - 12 - 2002
```

Description

Routine `TDATE` is used to retrieve today’s date. Obviously, the output is dependent upon the date the program is run.

NDAYS

This function computes the number of days from January 1, 1900, to the given date.

Function Return Value

NDAYS — Function value. (Output)
If `NDAYS` is negative, it indicates the number of days prior to January 1, 1900.

Required Arguments

IDAY — Day of the input date. (Input)

MONTH — Month of the input date. (Input)

IYEAR — Year of the input date. (Input)
1950 would correspond to the year 1950 A.D. and 50 would correspond to year 50 A.D.

FORTRAN 90 Interface

Generic: `NDAYS(IDAY, MONTH, IYEAR)`

Specific: The specific interface name is NDAYS.

FORTRAN 77 Interface

Single: NDAYS(IDAY, MONTH, IYEAR)

Example

The following example uses NDAYS to compute the number of days from January 15, 1986, to February 28, 1986:

```
      USE NDAYS_INT
      USE UMACH_INT
      INTEGER    IDAY, IYEAR, MONTH, NDAY0, NDAY1, NOUT
!
      IDAY  = 15
      MONTH = 1
      IYEAR = 1986
      NDAY0 = NDAYS(IDAY,MONTH,IYEAR)
      IDAY  = 28
      MONTH = 2
      IYEAR = 1986
      NDAY1 = NDAYS(IDAY,MONTH,IYEAR)
      CALL UMACH (2, NOUT)
      WRITE (NOUT,*) 'Number of days = ', NDAY1 - NDAY0
      END
```

Output

```
Number of days =   44
```

Comments

1. Informational error

Type	Code	
1	1	The Julian calendar, the first modern calendar, went into use in 45 B.C. No calendar prior to 45 B.C. was as universally used nor as accurate as the Julian. Therefore, it is assumed that the Julian calendar was in use prior to 45 B.C.

2. The number of days from one date to a second date can be computed by two references to NDAYS and then calculating the difference.

3. The beginning of the Gregorian calendar was the first day after October 4, 1582, which became October 15, 1582. Prior to that, the Julian calendar was in use. NDAYS makes the proper adjustment for the change in calendars.

Description

Function NDAYS returns the number of days from January 1, 1900, to the given date. The function NDAYS returns negative values for days prior to January 1, 1900. A negative IYEAR can be used to specify B.C. Input dates in year 0 and for October 5, 1582, through October 14, 1582, inclusive, do not exist; consequently, in these cases, NDAYS issues a terminal error.

NDYIN

Gives the date corresponding to the number of days since January 1, 1900.

Required Arguments

NDAYS — Number of days since January 1, 1900.   (Input)

IDAY — Day of the input date.   (Output)

MONTH — Month of the input date.   (Output)

IYEAR — Year of the input date.   (Output)
1950 would correspond to the year 195 A.D. and −50 would correspond to year 50 B.C.

FORTRAN 90 Interface

Generic: `CALL NDYIN (NDAYS, IDAY, MONTH, IYEAR)`

Specific: The specific interface name is NDYIN.

FORTRAN 77 Interface

Single: `CALL NDYIN (NDAYS, IDAY, MONTH, IYEAR)`

Example

The following example uses NDYIN to compute the date for the 100th day of 1986. This is accomplished by first using NDAYS (page 1634) to get the “day number” for December 31, 1985.

```
      USE NDYIN_INT
      USE NDAYS_INT
      USE UMACH_INT
      INTEGER    IDAY, IYEAR, MONTH, NDAYO, NOUT
!
      NDAY0 = NDAYS(31,12,1985)
      CALL NDYIN (NDAY0+100, IDAY, MONTH, IYEAR)
      CALL UMACH (2, NOUT)
      WRITE (NOUT,*) ’Day 100 of 1986 is (day-month-year) ’, IDAY, &
                   ’-’, MONTH, ’-’, IYEAR
      END
```

Output

```
Day 100 of 1986 is (day-month-year)   10-  4-  1986
```

Comments

The beginning of the Gregorian calendar was the first day after October 4, 1582, which became October 15, 1582. Prior to that, the Julian calendar was in use. Routine `NDYIN` makes the proper adjustment for the change in calendars.

Description

Routine `NDYIN` computes the date corresponding to the number of days since January 1, 1900. For an input value of `NDAYS` that is negative, the date
computed is prior to January 1, 1900. The routine `NDYIN` is the inverse of `NDAYS` (page 1634).

IDYWK

This function computes the day of the week for a given date.

Function Return Value

IDYWK — Function value. (Output)
The value of `IDYWK` ranges from 1 to 7, where 1 corresponds to Sunday and 7 corresponds to Saturday.

Required Arguments

IDAY — Day of the input date. (Input)

MONTH — Month of the input date. (Input)

IYEAR — Year of the input date. (Input)
1950 would correspond to the year 1950 A.D. and 50 would correspond to year 50 A.D.

FORTRAN 90 Interface

Generic: `IDYWK(IDAY, MONTH, IYEAR)`

Specific: The specific interface name is `IDYWK`.

FORTRAN 77 Interface

Single: `IDYWK(IDAY, MONTH, IYEAR)`

Example

The following example uses `IDYWK` to return the day of the week for February 24, 1963.

```
USE IDYWK_INT
USE UMACH_INT
INTEGER    IDAY, IYEAR, MONTH, NOUT
```

```
!
      IDAY  = 24
      MONTH = 2
      IYEAR = 1963
      CALL UMACH (2, NOUT)
      WRITE (NOUT,*) 'IDYWK (index for day of week) = ', &
                      IDYWK(IDAY,MONTH,IYEAR)
      END
```

Output

```
IDYWK (index for day of week) =   1
```

Comments

1. Informational error

Type	Code	
1	1	The Julian calendar, the first modern calendar, went into use in 45 B.C. No calendar prior to 45 B.C. was as universally used nor as accurate as the Julian. Therefore, it is assumed that the Julian calendar was in use prior to 45 B.C.

2. The beginning of the Gregorian calendar was the first day after October 4, 1582, which became October 15, 1582. Prior to that, the Julian calendar was in use. Function `IDYWK` makes the proper adjustment for the change in calendars.

Description

Function `IDYWK` returns an integer code that specifies the day of week for a given date. Sunday corresponds to 1, Monday corresponds to 2, and so forth.

A negative `IYEAR` can be used to specify B.C. Input dates in year 0 and for October 5, 1582, through October 14, 1582, inclusive, do not exist; consequently, in these cases, `IDYWK` issues a terminal error.

VERML

This function obtains IMSL MATH/LIBRARY-related version, system and serial numbers.

Function Return Value

VERML — `CHARACTER` string containing information. (Output)

Required Arguments

ISELCT — Option for the information to retrieve. (Input)

ISELCT	VERML
1	IMSL MATH/LIBRARY version number

2 Operating system (and version number) for which the library was produced.

3 Fortran compiler (and version number) for which the library was produced.

4 IMSL MATH/LIBRARY serial number

FORTRAN 90 Interface

Generic: `VERML(ISELCT)`

Specific: The specific interface name is `VERML`.

FORTRAN 77 Interface

Single: `VERML(ISELCT)`

Example

In this example, we print all of the information returned by `VERML` on a particular machine. The output is omitted because the results are system dependent.

```
      USE UMACH_INT
      USE VERML_INT
      INTEGER    ISELCT, NOUT
      CHARACTER  STRING(4)*50, TEMP*32
!
      STRING(1) = ’(’’ IMSL MATH/LIBRARY Version Number:  ’’, A)’
      STRING(2) = ’(’’ Operating System ID Number:  ’’, A)’
      STRING(3) = ’(’’ Fortran Compiler Version Number:  ’’, A)’
      STRING(4) = ’(’’ IMSL MATH/LIBRARY Serial Number:  ’’, A)’
!                                  Print the versions and numbers.
      CALL UMACH (2, NOUT)
      DO 10  ISELCT=1, 4
         TEMP = VERML(ISELCT)
         WRITE (NOUT,STRING(ISELCT)) TEMP
   10 CONTINUE
      END
```

Output

```
IMSL MATH/LIBRARY Version Number:  IMSL MATH/LIBRARY Version 2.0
Operating System ID Number:  SunOS 4.1.1
Fortran Compiler Version Number:  f77 Sun FORTRAN 1.3.1
IMSL MATH/LIBRARY Serial Number:  123456
```

RAND_GEN

Generates a rank-1 array of random numbers. The output array entries are positive and less than 1 in value.

Required Argument

X — Rank-1 array containing the random numbers. (Output)

Optional Arguments

`irnd = irnd` (Output)

Rank-1 integer array. These integers are the internal results of the Generalized Feedback Shift Register (GFSR) algorithm. The values are scaled to yield the floating-point array `X`. The output array entries are between 1 and $2^{31} - 1$ in value.

`istate_in = istate_in` (Input)

Rank-1 integer array of size 3p + 2, where p = 521, that defines the ensuing state of the GFSR generator. It is used to reset the internal tables to a previously defined state. It is the result of a previous use of the "istate_out=" optional argument.

`istate_out = istate_out` (Output)

Rank-1 integer array of size 3p + 2 that describes the current state of the GFSR generator. It is normally used to later reset the internal tables to the state defined following a return from the GFSR generator. It is the result of a use of the generator without a user initialization, or it is the result of a previous use of the optional argument "istate_in=" followed by updates to the internal tables from newly generated values. Example 2 illustrates use of istate_in and istate_`out` for setting and then resetting `rand_gen` so that the sequence of integers, `irnd`, is repeatable.

`iopt = iopt(:)` (Input[/Output])

Derived type array with the same precision as the array `x`; used for passing optional data to `rand_gen`. The options are as follows:

Packaged Options for `RAND_GEN`		
Option Prefix = ?	Option Name	Option Value
s_, d_	Rand_gen_generator_seed	1
s_, d_	Rand_gen_LCM_modulus	2
s_, d_	Rand_gen_use_Fushimi_start	3

`iopt(IO) = ?_options(?_rand_gen_generator_seed, ?_dummy)`

Sets the initial values for the GFSR. The present value of the seed, obtained by default from the real-time clock as described below, swaps places with `iopt(IO + 1)%idummy`. If the seed is set before any current usage of rand_gen, the exchanged value will be zero.

`iopt(IO) = ?_options(?_rand_gen_LCM_modulus, ?_dummy)`

`iopt(IO+1) = ?_options(modulus, ?_dummy)`

Sets the initial values for the GFSR. The present value of the LCM, with default value *k* = 16807, swaps places with `iopt(IO+1)%idummy`.

`iopt(IO) = ?_options(?_rand_gen_use_Fushimi_start, ?_dummy)`

Starts the GFSR sequence as suggested by Fushimi (1990). The default starting sequence is with the LCM recurrence described below.

FORTRAN 90 Interface

Generic: `CALL RAND_GEN` (X [,…])

Specific: The specific interface names are `S_RAND_GEN` and `D_RAND_GEN`.

Example 1: Running Mean and Variance

An array of random numbers is obtained. The sample mean and variance are computed. These values are compared with the same quantities computed using a stable method for the running means and variances, sequentially moving through the data. Details about the running mean and variance are found in Henrici (1982, pp. 21–23).

```
use rand_gen_int

      implicit none

! This is Example 1 for RAND_GEN.

      integer i
      integer, parameter :: n=1000
      real(kind(1e0)), parameter :: one=1e0, zero=0e0
      real(kind(1e0)) x(n), mean_1(0:n), mean_2(0:n), s_1(0:n), s_2(0:n)

! Obtain random numbers.
      call rand_gen(x)

! Calculate each partial mean.
      do i=1,n
        mean_1(i) = sum(x(1:i))/i
      end do

! Calculate each partial variance.
      do i=1,n
        s_1(i)=sum((x(1:i)-mean_1(i))**2)/i
      end do

      mean_2(0)=zero
      mean_2(1)=x(1)
      s_2(0:1)=zero

! Alternately calculate each running mean and variance,
! handling the random numbers once.
      do i=2,n
       mean_2(i)=((i-1)*mean_2(i-1)+x(i))/i
       s_2(i)   = (i-1)*s_2(i-1)/i+(mean_2(i)-x(i))**2/(i-1)
      end do
```

```
! Check that the two sets of means and variances agree.
      if (maxval(abs(mean_1(1:)-mean_2(1:))/mean_1(1:)) <= &
              sqrt(epsilon(one))) then
         if (maxval(abs(s_1(2:)-s_2(2:))/s_1(2:)) <= &
              sqrt(epsilon(one))) then
            write (*,*) 'Example 1 for RAND_GEN is correct.'
         end if
      end if

      end
```

Output

```
Example 1 for RAND_GEN is correct.
```

Description

This GFSR algorithm is based on the recurrence

$$x_t = x_{t-3p} \oplus x_{t-3p}$$

where $a \oplus b$ is the exclusive `OR` operation on two integers a and b. This operation is performed until `size(x)` numbers have been generated. The subscripts in the recurrence formula are computed modulo $3p$. These numbers are converted to floating point by effectively multiplying the positive integer quantity

$$x_t \cup 1$$

by a scale factor slightly smaller than 1./(*huge*(1)). The values $p = 521$ and $q = 32$ yield a sequence with a period approximately

$$2^p > 10^{156.8}$$

The default initial values for the sequence of integers $\{x_t\}$ are created by a congruential generator starting with an odd integer seed

$$m = v + |\, count \cap \left(2^{bit\_size(1)} - 1\right) | \cup 1$$

obtained by the Fortran 90 real-time clock routine:

```
CALL SYSTEM_CLOCK(COUNT=count,CLOCK_RATE=CLRATE)
```

An error condition is noted if the value of `CLRATE=0`. This indicates that the processor does not have a functioning real-time clock. In this exceptional case a starting seed must be provided by the user with the optional argument “`iopt=`” and option number `?_rand_generator_seed`. The value v is the current clock for this day, in milliseconds. This value is obtained using the date routine:

```
CALL DATE_AND_TIME(VALUES=values)
```

and converting `values(5:8)` to milliseconds.

The LCM generator initializes the sequence $\{x_t\}$ using the following recurrence:

$$m \leftarrow m \times k,\ \text{mod}\left(huge(1)/2\right)$$

The default value of k = 16807. Using the optional argument "`iopt=`" and the packaged option number `?_rand_gen_LCM_modulus`, k can be given an alternate value. The option number `?_rand_gen_generator_seed` can be used to set the initial value of m instead of using the asynchronous value given by the system clock. This is illustrated in Example 2. If the default choice of m results in an unsatisfactory starting sequence or it is necessary to duplicate the sequence, then it is recommended that users set the initial seed value to one of their own choosing. Resetting the seed complicates the usage of the routine.

This software is based on Fushimi (1990), who gives a more elaborate starting sequence for the {xt} . The starting sequence suggested by Fushimi can be used with the option number `?_rand_gen_use_Fushimi_start`. Fushimi's starting process is more expensive than the default method, and it is equivalent to starting in another place of the sequence with period 2^p.

Additional Examples

Example 2: Seeding, Using, and Restoring the Generator

```
     use rand_gen_int

      implicit none

! This is Example 2 for RAND_GEN.

      integer i
      integer, parameter :: n=34, p=521
      real(kind(1e0)), parameter :: one=1.0e0, zero=0.0e0
      integer irndi(n), i_out(3*p+2), hidden_message(n)
      real(kind(1e0)) x(n), y(n)
      type(s_options) :: iopti(2)=s_options(0,zero)
      character*34 message, returned_message

! This is the message to be hidden.
      message = 'SAVE YOURSELF.  WE ARE DISCOVERED!'

! Start the generator with a known seed.
      iopti(1) = s_options(s_rand_gen_generator_seed,zero)
      iopti(2) = s_options(123,zero)
      call rand_gen(x, iopt=iopti)

! Save the state of the generator.
      call rand_gen(x, istate_out=i_out)

! Get random integers.
      call rand_gen(y, irnd=irndi)

! Hide text using collating sequence subtracted from integers.
      do i=1, n
         hidden_message(i) = irndi(i) - ichar(message(i:i))
      end do

! Reset generator to previous state and generate the previous
! random integers.
      call rand_gen(x, irnd=irndi, istate_in=i_out)
```

```
! Subtract hidden text from integers and convert to character.
      do i=1, n
         returned_message(i:i) = char(irndi(i) - hidden_message(i))
      end do

! Check the results.
      if (returned_message == message) then



         write (*,*) 'Example 2 for RAND_GEN is correct.'
      end if

      end
```

Output

```
Example 2 for RAND_GEN is correct.
```

Example 3: Generating Strategy with a Histogram

We generate random integers but with the frequency as in a histogram with n_{bins} slots. The generator is initially used a large number of times to demonstrate that it is making choices with the same *shape* as the histogram. This is not required to generate samples. The program next generates a summary set of integers according to the histogram. These are not repeatable and are representative of the histogram in the sense of looking at 20 integers during generation of a *large number of samples.*

```
        use rand_gen_int
        use show_int

      implicit none

! This is Example 3 for RAND_GEN.

      integer i, i_bin, i_map, i_left, i_right
      integer, parameter :: n_work=1000
      integer, parameter :: n_bins=10
      integer, parameter :: scale=1000
      integer, parameter :: total_counts=100
      integer, parameter :: n_samples=total_counts*scale
      integer, dimension(n_bins) :: histogram=  &
        (/4,  6,  8, 14, 20, 17, 12,  9,  7,  3 /)
      integer, dimension(n_work) :: working=0
      integer, dimension(n_bins) :: distribution=0
      integer break_points(0:n_bins)
      real(kind(1e0)) rn(n_samples)
      real(kind(1e0)), parameter :: tolerance=0.005


      integer, parameter :: n_samples_20=20
      integer rand_num_20(n_samples_20)
      real(kind(1e0)) rn_20(n_samples_20)
```

```
! Compute the normalized cumulative distribution.
      break_points(0)=0
      do i=1,n_bins
        break_points(i)=break_points(i-1)+histogram(i)
      end do

      break_points=break_points*n_work/total_counts

! Obtain uniform random numbers.
        call rand_gen(rn)


! Set up the secondary mapping array.
      do i_bin=1,n_bins
        i_left=break_points(i_bin-1)+1
        i_right=break_points(i_bin)
        do i=i_left, i_right
          working(i)=i_bin
        end do
      end do

! Map the random numbers into the 'distribution' array.
! This is made approximately proportional to the histogram.
      do i=1,n_samples
        i_map=nint(rn(i)*(n_work-1)+1)
        distribution(working(i_map))=  &
          distribution(working(i_map))+1
      end do

! Check the agreement between the distribution of the
! generated random numbers and the original histogram.
       write (*, '(A)', advance='no') 'Original: '
       write (*, '(10I6)') histogram*scale
       write (*, '(A)', advance='no') 'Generated:'
       write (*, '(10I6)') distribution

      if (maxval(abs(histogram(1:)*scale-distribution(1:))) &
            <= tolerance*n_samples) then
        write(*, '(A/)') 'Example 3 for RAND_GEN is correct.'
      end if

! Generate 20 integers in 1, 10 according to the distribution
! induced by the histogram.
        call rand_gen(rn_20)

! Map from the uniform distribution to the induced distribution.
      do i=1,n_samples_20
        i_map=nint(rn_20(i)*(n_work-1)+1)
        rand_num_20(i)=working(i_map)
      end do

        call show(rand_num_20,&
'Twenty integers generated according to the histogram:')
      end
```

Output

```
Example 3 for RAND_GEN is correct.
```

Example 4: Generating with a Cosine Distribution

We generate random numbers based on the continuous distribution function

$$p(x) = \left(1 + \cos(x)\right)/2\pi, -\pi \le x \le \pi$$

Using the cumulative

$$q(x) = \int_{-\pi}^{x} p(t)\,dt = 1/2 + \left(x + \sin(x)\right)/2\pi$$

we generate the samples by obtaining uniform samples u, $0 < u < 1$ and solve the equation

$$q(x) - u = 0, -\pi < x < \pi$$

These are evaluated in vector form, that is all entries at one time, using Newton’s method:

$$x \leftarrow x - dx, dx = \left(q(x) - u\right)/p(x)$$

An iteration counter forces the loop to terminate, but this is not often required although it is an important detail.

```
      use rand_gen_int
      use show_int
      use Numerical_Libraries

        IMPLICIT NONE

! This is Example 4 for RAND_GEN.

      integer i, i_map, k
      integer, parameter :: n_bins=36
      integer, parameter :: offset=18
      integer, parameter :: n_samples=10000
      integer, parameter :: n_samples_30=30
      integer, parameter :: COUNT=15

      real(kind(1e0)) probabilities(n_bins)
      real(kind(1e0)), dimension(n_bins) :: counts=0.0
      real(kind(1e0)), dimension(n_samples) :: rn, x, f, fprime, dx
      real(kind(1e0)), dimension(n_samples_30) :: rn_30, &
               x_30, f_30, fprime_30, dx_30
      real(kind(1e0)), parameter :: one=1e0, zero=0e0, half=0.5e0
      real(kind(1e0)), parameter :: tolerance=0.01
      real(kind(1e0)) two_pi, omega

! Initialize values of 'two_pi' and 'omega'.
       two_pi=2.0*const((/'pi'/))
       omega=two_pi/n_bins

! Compute the probabilities for each bin according to
! the probability density (cos(x)+1)/(2*pi), -pi<x<pi.
      do i=1,n_bins
        probabilities(i)=(sin(omega*(i-offset))  &
            -sin(omega*(i-offset-1))+omega)/two_pi
      end do

! Obtain uniform random numbers in (0,1).
      call rand_gen(rn)

! Use Newton's method to solve the nonlinear equation:
! accumulated_distribution_function - random_number = 0.
      x=zero; k=0
      solve_equation: do
        f=(sin(x)+x)/two_pi+half-rn
        fprime=(one+cos(x))/two_pi
        dx=f/fprime
        x=x-dx; k=k+1
        if (maxval(abs(dx)) <= sqrt(epsilon(one)) &
              .or. k > COUNT) exit solve_equation
      end do solve_equation

! Map the random numbers 'x' array into the 'counts' array.
        do i=1,n_samples
          i_map=int(x(i)/omega+offset)+1
          counts(i_map)=counts(i_map)+one
        end do

! Normalize the counts array.
      counts=counts/n_samples

! Check that the generated random numbers are indeed
! based on the original distribution.
      if (maxval(abs(counts(1:)-probabilities(1:))) &
            <= tolerance) then
        write (*,'(a/)') 'Example 4 for RAND_GEN is correct.'
      end if

! Generate 30 random numbers in (-pi,pi) according to
! the probability density (cos(x)+1)/(2*pi), -pi<x<pi.
        call rand_gen(rn_30)

      x_30=0.0; k=0
      solve_equation_30: do
        f_30=(sin(x_30)+x_30)/two_pi+half-rn_30
        fprime_30=(one+cos(x_30))/two_pi
        dx_30=f_30/fprime_30
        x_30=x_30-dx_30
        if (maxval(abs(dx_30)) <= sqrt(epsilon(one))&
             .or. k > COUNT) exit solve_equation_30
      end do solve_equation_30

        write(*,'(A)') 'Thirty random numbers generated ', &
                   'according to the probability density ',&
                   'pdf(x)=(cos(x)+1)/(2*pi), -pi<x<pi:'
        call show(x_30)
        end
```

Output

```
Example 4 for RAND_GEN is correct.
```

Fatal and Terminal Error Messages

See the *messages.gls* file for error messages for rand_gen. These error messages are numbered 521–528; 541–548.

RNGET

Retrieves the current value of the seed used in the IMSL random number generators.

Required Arguments

ISEED — The seed of the random number generator. (Output)
ISEED is in the range (1, 2147483646).

FORTRAN 90 Interface

Generic: CALL RNGET (ISEED)

Specific: The specific interface name is RNGET.

FORTRAN 77 Interface

Single: CALL RNGET (ISEED)

Example

The following FORTRAN statements illustrate the use of RNGET:

```
      INTEGER ISEED
!                          Call RNSET to initialize the seed.
      CALL RNSET(123457)
!                          Do some simulations.
         ...
         ...
      CALL RNGET(ISEED)
!                          Save ISEED.  If the simulation is to be continued
!                          in a different program, ISEED should be output,
!                          possibly to a file.
         ...
         ...
```

```
!                        When the simulations begun above are to be
!                        restarted, restore ISEED to the value obtained
!                        above and use as input to RNSET.
      CALL RNSET(ISEED)
!                        Now continue the simulations.
         ...
         ...
```

Description

Routine RNGET retrieves the current value of the "seed" used in the IMSL random number generators. A reason for doing this would be to restart a simulation, using RNSET to reset the seed.

RNSET

Initializes a random seed for use in the IMSL random number generators.

Required Arguments

ISEED — The seed of the random number generator. (Input)
ISEED must be in the range (0, 2147483646). If ISEED is zero, a value is computed using the system clock; and, hence, the results of programs using the IMSL random number generators will be different at different times.

FORTRAN 90 Interface

Generic: CALL RNSET (ISEED)

Specific: The specific interface name is RNSET .

FORTRAN 77 Interface

Single: CALL RNSET (ISEED)

Example

The following FORTRAN statements illustrate the use of RNSET:

```
      INTEGER ISEED
!                        Call RNSET to initialize the seed via the
!                        system clock.
      CALL RNSET(0)
!                        Do some simulations.
         ...
         ...
!                        Obtain the current value of the seed.
```

```
        CALL RNGET(ISEED)
!                          If the simulation is to be continued in a
!                          different program, ISEED should be output,
!                          possibly to a file.
             ...
             ...
!                          When the simulations begun above are to be
!                          restarted, restore ISEED to the value
!                          obtained above, and use as input to RNSET.
        CALL RNSET(ISEED)
!                          Now continue the simulations.
             ...
             ...
```

Description

Routine RNSET is used to initialize the seed used in the IMSL random number generators. If the seed is not initialized prior to invocation of any of the routines for random number generation by calling RNSET, the seed is initialized via the system clock. The seed can be reinitialized to a clock-dependent value by calling RNSET with ISEED set to 0.

The effect of RNSET is to set some values in a FORTRAN COMMON block that is used by the random number generators.

A common use of RNSET is in conjunction with RNGET (page 1648) to restart a simulation.

RNOPT

Selects the uniform (0, 1) multiplicative congruential pseudorandom number generator.

Required Arguments

IOPT — Indicator of the generator. (Input)
The random number generator is either a multiplicative congruential generator with modulus $2^{31} - 1$ or a GFSR generator. IOPT is used to choose the multiplier and whether or not shuffling is done, or else to choose the GFSR method.

IOPT	**Generator**
1	The multiplier 16807 is used.
2	The multiplier 16807 is used with shuffling.
3	The multiplier 397204094 is used.
4	The multiplier 397204094 is used with shuffling.
5	The multiplier 950706376 is used.
6	The multiplier 950706376 is used with shuffling.

7 GFSR, with the recursion $X_t = X_{t-1563} \oplus X_{t-96}$ is used.

FORTRAN 90 Interface

Generic: `CALL RNOPT (IOPT)`

Specific: The specific interface name is `RNOPT`.

FORTRAN 77 Interface

Single: `CALL RNOPT (IOPT)`

Description

The IMSL uniform pseudorandom number generators use a multiplicative congruential method, with or without shuffling or else a GFSR method. Routine `RNOPT` determines which method is used; and in the case of a multiplicative congruential method, it determines the value of the multiplier and whether or not to use shuffling. The description of `RNUN` (page 1653) may provide some guidance in the choice of the form of the generator. If no selection is made explicitly, the generators use the multiplier 16807 without shuffling. This form of the generator has been in use for some time (see Lewis, Goodman, and Miller, 1969). This is the generator formerly known as `GGUBS` in the IMSL Library. It is the “minimal standard generator” discussed by Park and Miller (1988).

Example

The FORTRAN statement

```
            CALL RNOPT(1)
```

would select the simple multiplicative congruential generator with multiplier 16807. Since this is the same as the default, this statement would have no effect unless `RNOPT` had previously been called in the same program to select a different generator.

RNUNF

This function generates a pseudorandom number from a uniform (0, 1) distribution.

Function Return Value

RNUNF — Function value, a random uniform (0, 1) deviate. (Output)
See Comment 1.

Required Arguments

None

FORTRAN 90 Interface

Generic: `RNUNF ()`

Specific: The specific interface names are `S_RNUNF` and `D_RNUNF`.

FORTRAN 77 Interface

Single: `RNUNF ()`

Double: The double precision name is `DRNUNF`.

Example

In this example, `RNUNF` is used to generate five pseudorandom uniform numbers. Since `RNOPT` (page 1650) is not called, the generator used is a simple multiplicative congruential one with a multiplier of 16807.

```
      USE RNUNF_INT
      USE RNSET_INT
      USE UMACH_INT
      INTEGER    I, ISEED, NOUT
      REAL       R(5)
!
      CALL UMACH (2, NOUT)
      ISEED = 123457
      CALL RNSET (ISEED)
      DO 10  I=1, 5
         R(I) = RNUNF()
   10 CONTINUE
      WRITE (NOUT,99999) R
99999 FORMAT ('      Uniform random deviates: ', 5F8.4)
      END
```

Output

```
Uniform random deviates:   0.9662  0.2607  0.7663  0.5693  0.8448
```

Comments

1. If the generic version of this function is used, the immediate result must be stored in a variable before use in an expression. For example:

   ```
   X = RNUNF(6)
   Y = SQRT(X)
   ```

   must be used rather than

   ```
   Y = SQRT(RNUNF(6))
   ```

   If this is too much of a restriction on the programmer, then the specific name can be used without this restriction.

2. Routine RNSET (page 1649) can be used to initialize the seed of the random number generator. The routine RNOPT (page 1650) can be used to select the form of the generator.

3. This function has a side effect: it changes the value of the seed, which is passed through a common block.

Description

Routine RNUNF is the function form of RNUN (page 1653). The routine RNUNF generates pseudorandom numbers from a uniform (0, 1) distribution. The algorithm used is determined by RNOPT (page 1650). The values returned by RNUNF are positive and less than 1.0.

If several uniform deviates are needed, it may be more efficient to obtain them all at once by a call to RNUN rather than by several references to RNUNF.

RNUN

Generates pseudorandom numbers from a uniform (0, 1) distribution.

Required Arguments

R — Vector of length NR containing the random uniform (0, 1) deviates. (Output)

Optional Arguments

NR — Number of random numbers to generate. (Input)
Default: NR = size (R,1).

FORTRAN 90 Interface

Generic: CALL RNUN (R [,…])

Specific: The specific interface names are S_RNUN and D_RNUN.

FORTRAN 77 Interface

Single: CALL RNUN (NR, R)

Double: The double precision name is DRNUN.

Example

In this example, RNUN is used to generate five pseudorandom uniform numbers. Since RNOPT (page 1650) is not called, the generator used is a simple multiplicative congruential one with a multiplier of 16807.

```
      USE RNUN_INT
      USE RNSET_INT
      USE UMACH_INT
      INTEGER    ISEED, NOUT, NR
      REAL       R(5)
!
      CALL UMACH (2, NOUT)
      NR    = 5
      ISEED = 123457
      CALL RNSET (ISEED)
      CALL RNUN (R)
      WRITE (NOUT,99999) R
99999 FORMAT (’      Uniform random deviates: ’, 5F8.4)
      END
```

Output

```
Uniform random deviates:    .9662   .2607   .7663   .5693   .8448
```

Comments

The routine RNSET (page 1649) can be used to initialize the seed of the random number generator. The routine RNOPT (page 1650) can be used to select the form of the generator.

Description

Routine RNUN generates pseudorandom numbers from a uniform (0,1) distribution using either a multiplicative congruential method or a generalized feedback shift register (GFSR) method. The form of the multiplicative congruential generator is

$$x_i \equiv cx_{i-1} \bmod \left(2^{31}-1\right)$$

Each x_i is then scaled into the unit interval (0,1). The possible values for c in the IMSL generators are 16807, 397204094, and 950706376. The selection is made by the routine RNOPT (page 1650). The choice of 16807 will result in the fastest execution time. If no selection is made explicitly, the routines use the multiplier 16807.

The user can also select a shuffled version of the multiplicative congruential generators. In this scheme, a table is filled with the first 128 uniform (0,1) numbers resulting from the simple multiplicative congruential generator. Then, for each x_i from the simple generator, the low-order bits of x_i are used to select a random integer, j, from 1 to 128. The j-th entry in the table is then delivered as the random number; and x_i, after being scaled into the unit interval, is inserted into the j-th position in the table.

The GFSR method is based on the recursion $X_t = X_{t-1563} \oplus X_{t-96}$. This generator, which is different from earlier GFSR generators, was proposed by Fushimi (1990), who discusses the theory behind the generator and reports on several empirical tests of it. The values returned in R by RNUN are positive and less than 1.0. Values in R may be smaller than the smallest relative spacing, however. Hence, it may be the case that some value R(i) is such that 1.0 − R(i) = 1.0.

Deviates from the distribution with uniform density over the interval (A, B) can be obtained by scaling the output from RNUN. The following statements (in single precision) would yield random deviates from a uniform (A, B) distribution:

```
          CALL RNUN (NR, R)
          CALL SSCAL (NR, B-A, R, 1)
          CALL SADD (NR, A, R, 1)
```

FAURE_INIT

Shuffled Faure sequence initialization.

Required Arguments

NDIM — The dimension of the hyper-rectangle. (Input)

STATE — An `IMSL_FAURE` pointer for the derived type created by the call to `FAURE_INIT`. The output contains information about the sequence. Use `?_IMSL_FAURE` as the type, where `?_` is `S_` or `D_` depending on precision. (Output)

Optional Arguments

NBASE — The base of the Faure sequence. (Input)

Default: The smallest prime number greater than or equal to `NDIM`.

NSKIP — The number of points to be skipped at the beginning of the Faure sequence. (Input)

Default: $\lfloor \texttt{base}^{m/2-1} \rfloor$, where $m = \lfloor \log B / \log \texttt{base} \rfloor$ and B is the largest machine representable integer.

FORTRAN 90 Interface

Generic: `CALL FAURE_INIT (NDIM, STATE [,…])`

Specific: The specific interface names are `S_FAURE_INIT` and `D_FAURE_INIT`.

FAURE_FREE

Frees the structure containing information about the Faure sequence.

Required Arguments

STATE — An `IMSL_FAURE` pointer containing the structure created by the call to `FAURE_INIT`. (Input/Output)

FORTRAN 90 Interface

Generic: `CALL FAURE_FREE (STATE)`

Specific: The specific interface names are S_FAURE_FREE and D_FAURE_FREE.

FAURE_NEXT

Computes a shuffled Faure sequence.

Required Arguments

STATE — An IMSL_FAURE pointer containing the structure created by the call to FAURE_INIT. The structure contains information about the sequence. The structure should be freed using FAURE_FREE after it is no longer needed. (Input/Output)

NEXT_PT — Vector of length NDIM containing the next point in the shuffled Faure sequence, where NDIM is the dimension of the hyper-rectangle specified in FAURE_INIT. (Output)

Optional Arguments

IMSL_RETURN_SKIP — Returns the current point in the sequence. The sequence can be restarted by calling FAURE_INIT using this value for NSKIP, and using the same value for NDIM. (Input)

FORTRAN 90 Interface

Generic: CALL FAURE_NEXT (STATE, NEXT_PT [,…])

Specific: The specific interface names are S_FAURE_NEXT and D_FAURE_NEXT.

Example

In this example, five points in the Faure sequence are computed. The points are in the three-dimensional unit cube.

Note that FAURE_INIT is used to create a structure that holds the state of the sequence. Each call to FAURE_NEXT returns the next point in the sequence and updates the IMSL_FAURE structure. The final call to FAURE_FREE frees data items, stored in the structure, that were allocated by FAURE_INIT.

```
      use faure_int
      implicit none
      type (s_imsl_faure), pointer  :: state
      real(kind(1e0))           :: x(3)
      integer,parameter :: ndim=3
      integer           :: k
!                                  CREATE THE STRUCTURE THAT HOLDS
!                                  THE STATE OF THE SEQUENCE.
      call faure_init(ndim, state)
!                                  GET THE NEXT POINT IN THE SEQUENCE
```

```
      do k=1,5
         call faure_next(state, x)
         write(*,'(3F15.3)') x(1), x(2) , x(3)
      enddo
!                                  FREE DATA ITEMS STORED IN
!                                  state STRUCTURE
      call faure_free(state)
     end
```

Output

```
  0.334      0.493      0.064
  0.667      0.826      0.397
  0.778      0.270      0.175
  0.111      0.604      0.509
  0.445      0.937      0.842
```

Description

The routines `FAURE_INT` and `FAURE_NEXT` are used to generate shuffled Faure sequence of low discrepancy *n*-dimensional points. Low discrepency series fill an *n*-dimensional cube more uniformly than psuedo-random sequences, and are used in multivariate quadrature, simulation, and global optimization. Because of this uniformity, use of low discrepency series is generally more effiicient than psuedo-random series for multivariate Monte Carlo methods. See the IMSL routine `QMC` (Chapter 4, Integration and Differentiation) for a discussion of quasi-Monte Carlo quadrature based on low discrepancy series.

Discrepancy measures the deviation from uniformity of a point set.

The discrepancy of the point set $x_1,...,x_n \in [0,1]^d$, $d \geq 1$, is defined

$$D_n^{(d)} = \sup_E \left| \frac{A(E;n)}{n} - \lambda(E) \right|,$$

where the supremum is over all subsets of $[0, 1]^d$ of the form

$$E = \left[0, t_1\right) \times ... \times \left[0, t_d\right),\ 0 \leq t_j \leq 1,\ 1 \leq j \leq d\ ,$$

λ is the Lebesque measure, and $A(E;n)$ is the number of the x_j contained in *E*.

The sequence x_1, x_2, … of points $[0,1]^d$ is a low-discrepancy sequence if there exists a constant *c*(*d*), depending only on *d*, such that

$$D_n^{(d)} \leq c(d) \frac{(\log n)^d}{n}$$

for all *n*>1.

Generalized Faure sequences can be defined for any prime base *b*≥*d*. The lowest bound for the discrepancy is obtained for the smallest prime *b*≥*d*, so the optional argument `NBASE` defaults to the smallest prime greater than or equal to the dimension.

The generalized Faure sequence x_1, x_2, …, is computed as follows:

Write the positive integer *n* in its *b*-ary expansion,

$$n = \sum_{i=0}^{\infty} a_i(n) b^i$$

where $a_i(n)$ are integers, $0 \le a_i(n) < b$.

The *j*-th coordinate of x_n is

$$x_n^{(j)} = \sum_{k=0}^{\infty}\sum_{d=0}^{\infty} c_{kd}^{(j)}\ a_d(n)\ b^{-k-1}, \qquad 1 \le j \le d$$

The generator matrix for the series, $c_{k\,d}^{(j)}$, is defined to be

$$c_{kd}^{(j)} = j^{\,d-k} c_{k\,d}$$

and $c_{k\,d}$ is an element of the Pascal matrix,

$$c_{k\,d} = \begin{cases} \dfrac{d!}{c!(d-c)!} & k \le d \\ 0 & k > d \end{cases}$$

It is faster to compute a shuffled Faure sequence than to compute the Faure sequence itself. It can be shown that this shuffling preserves the low-discrepancy property.

The shuffling used is the *b*-ary Gray code. The function *G*(*n*) maps the positive integer *n* into the integer given by its *b*-ary expansion.

The sequence computed by this function is *x*(*G*(*n*)), where *x* is the generalized Faure sequence.

IUMAG

This routine handles MATH/LIBRARY and STAT/LIBRARY type `INTEGER` options.

Required Arguments

PRODNM — Product name. Use either “`MATH`” or “`STAT`.” (Input)

ICHP — Chapter number of the routine that uses the options. (Input)

IACT — 1 if user desires to “get” or read options, or 2 if user desires to “put” or write options. (Input)

NUMOPT — Size of `IOPTS`. (Input)

IOPTS — Integer array of size `NUMOPT` containing the option numbers to “get” or “put.” (Input)

IVALS — Integer array containing the option values. These values are arrays corresponding to the individual options in IOPTS in sequential order. The size of IVALS is the sum of the sizes of the individual options.   (Input/Output)

FORTRAN 90 Interface

Generic: CALL IUMAG (PRODNM, ICHP, IACT, NUMOPT, IOPTS, IVALS)

Specific: The specific interface name is IUMAG.

FORTRAN 77 Interface

Single: CALL IUMAG (PRODNM, ICHP, IACT, NUMOPT, IOPTS, IVALS)

Example

The number of iterations allowed for the constrained least squares solver LCLSQ that calls L2LSQ is changed from the default value of max(*nra*, *nca*) to the value 6. The default value is restored after the call to LCLSQ. This change has no effect on the solution. It is used only for illustration. The first two arguments required for the call to IUMAG are defined by the product name, “MATH,” and chapter number, 1, where LCLSQ is documented. The argument IACT denotes a write or “put” operation. There is one option to change so NUMOPT has the value 1. The arguments for the option number, 14, and the new value, 6, are defined by reading the documentation for LCLSQ.

```
      USE IUMAG_INT
      USE LCLSQ_INT
      USE UMACH_INT
      USE SNRM2_INT
!
!     Solve the following in the least squares sense:
!           3x1 + 2x2 +  x3 = 3.3
!           4x1 + 2x2 +  x3 = 2.3
!           2x1 + 2x2 +  x3 = 1.3
!            x1 +  x2 +  x3 = 1.0
!
!     Subject to:  x1 + x2 + x3 <= 1
!                  0 <= x1 <= .5
!                  0 <= x2 <= .5
!                  0 <= x3 <= .5
!
! ----------------------------------------------------------------------
!                                Declaration of variables
!
      INTEGER    ICHP, IPUT, LDA, LDC, MCON, NCA, NEWMAX, NRA, NUMOPT
      PARAMETER  (ICHP=1, IPUT=2, MCON=1, NCA=3, NEWMAX=14, NRA=4, &
                NUMOPT=1, LDA=NRA, LDC=MCON)
!
      INTEGER    IOPT(1), IRTYPE(MCON), IVAL(1), NOUT
      REAL       A(LDA,NCA), B(NRA), BC(MCON), C(LDC,NCA), RES(NRA), &
                RESNRM, XLB(NCA), XSOL(NCA), XUB(NCA)
```

```
!                                 Data initialization
!
      DATA A/3.0E0, 4.0E0, 2.0E0, 1.0E0, 2.0E0, 2.0E0, 2.0E0, 1.0E0, &
          1.0E0, 1.0E0, 1.0E0, 1.0E0/, B/3.3E0, 2.3E0, 1.3E0, 1.0E0/, &
          C/3*1.0E0/, BC/1.0E0/, IRTYPE/1/, XLB/3*0.0E0/, XUB/3*.5E0/
! ----------------------------------------------------------------------
!
!                                 Reset the maximum number of

!                                 iterations to use in the solver.
!                                 The value 14 is the option number.
!                                 The value 6 is the new maximum.
      IOPT(1) = NEWMAX
      IVAL(1) = 6
      CALL IUMAG (’math’, ICHP, IPUT, NUMOPT, IOPT, IVAL)
!                                 -------------------------------------
!                                 ---------------------------------
!
!                                 Solve the bounded, constrained
!                                 least squares problem.
!
      CALL LCLSQ (A, B, C, BC, B, IRTYPE, XLB, XUB, XSOL, RES=RES)

!                                 Compute the 2-norm of the residuals.
      RESNRM = SNRM2(NRA,RES,1)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) XSOL, RES, RESNRM
!                                 -------------------------------------
!                                 ---------------------------------
!                                 Reset the maximum number of
!                                 iterations to its default value.
!                                 This is not required but is
!                                 recommended programming practice.
      IOPT(1) = -IOPT(1)
      CALL IUMAG (’math’, ICHP, IPUT, NUMOPT, IOPT, IVAL)
!                                 -------------------------------------
!                                 ---------------------------------
!
99999 FORMAT (’  The solution is ’, 3F9.4, //, ’  The residuals ’, &
             ’evaluated at the solution are ’, /, 18X, 4F9.4, //, &
             ’  The norm of the residual vector is ’, F8.4)
!
      END
```

Output

```
The solution is    0.5000   0.3000   0.2000

The residuals evaluated at the solution are
-1.0000   0.5000   0.5000   0.0000

The norm of the residual vector is   1.2247
```

Comments

1. Users can normally avoid reading about options when first using a routine that calls IUMAG.

2. Let I be any value between 1 and NUMOPT. A negative value of IOPTS(I) refers to option number −IOPTS(I) but with a different effect: For a "get" operation, the default values are returned in IVALS. For a "put" operation, the default values replace the current values. In the case of a "put," entries of IVALS are not allocated by the user and are not used by IUMAG.

3. Both positive and negative values of IOPTS can be used.

4. INTEGER Options

   1 If the value is positive, print the next activity for any library routine that uses the Options Manager codes IUMAG, SUMAG, or DUMAG. Each printing step decrements the value if it is positive.
   Default value is 0.

   2 If the value is 2, perform error checking in IUMAG (page 1658), SUMAG (page 1661), and DUMAG (page 1664) such as the verifying of valid option numbers and the validity of input data. If the value is 1, do not perform error checking.
   Default value is 2.

   3 This value is used for testing the installation of IUMAG by other IMSL software.
   Default value is 3.

Description

The Options Manager routine IUMAG reads or writes INTEGER data for some MATH/LIBRARY and STAT/LIBRARY codes. See Atchison and Hanson (1991) for more complete details.

There are MATH/LIBRARY routines in Chapters 1, 2, and 5 that now use IUMAG to communicate optional data from the user.

UMAG

This routine handles MATH/LIBRARY and STAT/LIBRARY type REAL and double precision options.

Required Arguments

PRODNM — Product name. Use either "MATH" or "STAT." (Input)

ICHP — Chapter number of the routine that uses the options. (Input)

IACT — 1 if user desires to "get" or read options, or 2 if user desires to "put" or write options. (Input)

IOPTS — Integer array of size `NUMOPT` containing the option numbers to “get” or “put.” (Input)

SVALS — Array containing the option values. These values are arrays corresponding to the individual options in `IOPTS` in sequential order. The size of `SVALS` is the sum of the sizes of the individual options. (Input/Output)

Optional Arguments

NUMOPT — Size of `IOPTS`. (Input)
Default: `NUMOPT` = size (`IOPTS`,1).

FORTRAN 90 Interface

Generic: `CALL UMAG (PRODNM, ICHP, IACT, IOPTS, SVALS [,…])`

Specific: The specific interface names are `S_UMAG` and `D_UMAG`.

FORTRAN 77 Interface

Single: `CALL SUMAG (PRODNM, ICHP, IACT, NUMOPT, IOPTS, SVALS)`

Double: The double precision name is `DUMAG`.

Example

The rank determination tolerance for the constrained least squares solver `LCLSQ` that calls `L2LSQ` is changed from the default value of `SQRT`(`AMACH`(4)) to the value 0.01. The default value is restored after the call to `LCLSQ`. This change has no effect on the solution. It is used only for illustration. The first two arguments required for the call to `SUMAG` are defined by the product name, “MATH,” and chapter number, 1, where `LCLSQ` is documented. The argument `IACT` denotes a write or “put” operation. There is one option to change so `NUMOPT` has the value 1. The arguments for the option number, 2, and the new value, 0.01E+0, are defined by reading the documentation for `LCLSQ`.

```
      USE UMAG_INT
      USE LCLSQ_INT
      USE UMACH_INT
      USE SNRM2_INT
!
!     Solve the following in the least squares sense:
!           3x1 + 2x2 +  x3 = 3.3
!           4x1 + 2x2 +  x3 = 2.3
!           2x1 + 2x2 +  x3 = 1.3
!            x1 +  x2 +  x3 = 1.0
!
!     Subject to:  x1 + x2 + x3 <= 1
!                  0 <= x1 <= .5
!                  0 <= x2 <= .5
!                  0 <= x3 <= .5
```

```
!
! -----------------------------------------------------------------------
!                                 Declaration of variables
!
      INTEGER    ICHP, IPUT, LDA, LDC, MCON, NCA, NEWTOL, NRA, NUMOPT
      PARAMETER  (ICHP=1, IPUT=2, MCON=1, NCA=3, NEWTOL=2, NRA=4, &
                NUMOPT=1, LDA=NRA, LDC=MCON)
!
      INTEGER    IOPT(1), IRTYPE(MCON), NOUT
      REAL       A(LDA,NCA), B(NRA), BC(MCON), C(LDC,NCA), RES(NRA), &
                RESNRM, SVAL(1), XLB(NCA), XSOL(NCA), XUB(NCA)
!                                 Data initialization
!
      DATA A/3.0E0, 4.0E0, 2.0E0, 1.0E0, 2.0E0, 2.0E0, 2.0E0, 1.0E0, &
          1.0E0, 1.0E0, 1.0E0, 1.0E0/, B/3.3E0, 2.3E0, 1.3E0, 1.0E0/, &
          C/3*1.0E0/, BC/1.0E0/, IRTYPE/1/, XLB/3*0.0E0/, XUB/3*.5E0/
! -----------------------------------------------------------------------
!
!                                 Reset the rank determination
!                                 tolerance used in the solver.
!                                 The value 2 is the option number.
!                                 The value 0.01 is the new tolerance.
!
      IOPT(1) = NEWTOL
      SVAL(1) = 0.01E+0
      CALL UMAG (’math’, ICHP, IPUT, IOPT, SVAL)
!                                 -------------------------------------
!                                 ---------------------------------
!
!                                 Solve the bounded, constrained
!                                 least squares problem.
!
      CALL LCLSQ (A, B, C, BC, BC, IRTYPE, XLB, XUB, XSOL, RES=RES)
!                                 Compute the 2-norm of the residuals.
      RESNRM = SNRM2(NRA,RES,1)
!                                 Print results
      CALL UMACH (2, NOUT)
      WRITE (NOUT,99999) XSOL, RES, RESNRM
!                                 -------------------------------------
!                                 ---------------------------------

!                                 Reset the rank determination
!                                 tolerance to its default value.
!                                 This is not required but is
!                                 recommended programming practice.
      IOPT(1) = -IOPT(1)
      CALL UMAG (’math’, ICHP, IPUT, IOPT, SVAL)
!                                 -------------------------------------
!                                 ---------------------------------
!
99999 FORMAT (’  The solution is ’, 3F9.4, //, ’  The residuals ’, &
            ’evaluated at the solution are ’, /, 18X, 4F9.4, //, &
            ’  The norm of the residual vector is ’, F8.4)
!
      END
```

Output

```
The solution is    0.5000   0.3000   0.2000

The residuals evaluated at the solution are
-1.0000   0.5000   0.5000   0.0000

The norm of the residual vector is   1.2247
```

Comments

1. Users can normally avoid reading about options when first using a routine that calls `SUMAG`.

2. Let `I` be any value between 1 and `NUMOPT`. A negative value of `IOPTS(I)` refers to option number −`IOPTS(I)` but with a different effect: For a "get" operation, the default values are returned in `SVALS`. For a "put" operation, the default values replace the current values. In the case of a "put," entries of `SVALS` are not allocated by the user and are not used by `SUMAG`.

3. Both positive and negative values of `IOPTS` can be used.

4. Floating Point Options

   1 This value is used for testing the installation of `SUMAG` by other IMSL software. Default value is `3.0E0`.

Description

The Options Manager routine `SUMAG` reads or writes `REAL` data for some MATH/LIBRARY and STAT/LIBRARY codes. See Atchison and Hanson (1991) for more complete details. There are MATH/LIBRARY routines in Chapters 1 and 5 that now use `SUMAG` to communicate optional data from the user.

SUMAG/DUMAG

See `UMAG`.

PLOTP

Prints a plot of up to 10 sets of points.

Required Arguments

X — Vector of length `NDATA` containing the values of the independent variable. (Input)

A — Matrix of dimension `NDATA` by `NFUN` containing the `NFUN` sets of dependent variable values. (Input)

SYMBOL — CHARACTER string of length NFUN. (Input)
SYMBOL(I : I) is the symbol used to plot function I.

XTITLE — CHARACTER string used to label the *x*-axis. (Input)

YTITLE — CHARACTER string used to label the *y*-axis. (Input)

TITLE — CHARACTER string used to label the plot. (Input)

Optional Arguments

NDATA — Number of independent variable data points. (Input)
Default: NDATA = size (X,1).

NFUN — Number of sets of points. (Input)
NFUN must be less than or equal to 10.
Default: NFUN = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Default: LDA = size (A, 1).

INC — Increment between elements of the data to be used. (Input)
PLOTP plots X(1 + (I – 1) * INC) for I = 1, 2, …, NDATA.
Default: INC = 1.

RANGE — Vector of length four specifying minimum *x*, maximum *x*, minimum *y* and maximum *y*. (Input)
PLOTP will calculate the range of the axis if the minimum and maximum of that range are equal.
Default: RANGE = 1.e0.

FORTRAN 90 Interface

Generic: `CALL PLOTP (X, A, SYMBOL, XTITLE, YTITLE, TITLE [,…])`

Specific: The specific interface names are S_PLOTP and D_PLOTP.

FORTRAN 77 Interface

Single:
```
CALL PLOTP (NDATA, NFUN, X, A, LDA, INC, RANGE, SYMBOL,
            XTITLE, YTITLE, TITLE)
```

Double: The double precision name is DPLOTP.

Example

This example plots the sine and cosine functions from – 3.5 to + 3.5 and sets page width and length to 78 and 40, respectively, by calling PGOPT (page 1599) in advance.

```
      USE PLOTP_INT
      USE CONST_INT
      USE PGOPT_INT
      INTEGER    I, IPAGE
      REAL       A(200,2), DELX, PI, RANGE(4), X(200)
      CHARACTER  SYMBOL*2
      INTRINSIC  COS, SIN
!
      DATA SYMBOL/'SC'/
      DATA RANGE/-3.5, 3.5, -1.2, 1.2/
!
      PI     = 3.14159
      DELX   = 2.*PI/199.
      DO 10  I= 1, 200
         X(I)   = -PI + FLOAT(I-1) * DELX
         A(I,1) = SIN(X(I))
         A(I,2) = COS(X(I))
   10 CONTINUE
!                                 Set page width and length
      IPAGE = 78
      CALL PGOPT (-1, IPAGE)
      IPAGE = 40
      CALL PGOPT (-2, IPAGE)
      CALL PLOTP (X, A, SYMBOL, 'X AXIS', 'Y AXIS', ' C = COS,   S = SIN', &
      RANGE=RANGE)
!
      END
```

Output

```
                         C = COS,   S = SIN

         1.2 ::::+:::::::::::::::+:::::::::::::::+:::::::::::::::+::::
             .                           I                           .
             .                           I                           .
             .                        CCCCCCC     SSSSSSSS           .
             .                       CC  I  CC   SS      SS          .
        0.8 +                      C    I    C SS        SS         +
             .                     C     I     MS          SS        .
             .                    C      I    SSC           SS       .
             .                   CC      I   SS CC           SS      .
             .                  CC       I   S   CC           S      .
        0.4 +                  C        I  S     C            S     +
             .                 C         I SS      C           SS    .
Y            .                CC         I S       CC           S    .
             .                C          IS         C            S   .
A            .               C           SS          C           SS  .
X        0.0 +--S-----------CC-----------S-----------CC-----------S--+
I            .  SS         CC           SS            CC             .
S            .   S         C            SI             C             .
             .    S       CC           S I             CC            .
             .    SS      C           SS I              C            .
        -0.4 +     S     C            S  I               C           +
             .      S   CC           S   I               CC          .
             .      SS CC           SS   I                CC         .
             .       SSC           SS    I                 C         .
             .        MS          SS     I                  C        .
        -0.8 +       C SS        SS      I                   C       +
             .     CC   SS      SS       I                    CC     .
             .  CCCC     SSSSSSSS        I                     CCCC  .
             .  C                        I                        C  .
             .                           I                           .
        -1.2 ::::+:::::::::::::::+:::::::::::::::+:::::::::::::::+::::
                -3              -1               1               3

                                      X AXIS
```

Comments

1. Informational errors

Type	Code	
3	7	NFUN is greater than 10. Only the first 10 functions are plotted.
3	8	TITLE is too long. TITLE is truncated from the right side.
3	9	YTITLE is too long. YTITLE is truncated from the right side.
3	10	XTITLE is too long. XTITLE is truncated from the right side. The maximum number of characters allowed depends on the page width and the page length. See Comment 5 below for more information.

2. YTITLE and TITLE are automatically centered.

3. For multiple plots, the character M is used if the same print position is shared by two or more data sets.

4. Output is written to the unit specified by UMACH (see Reference Material).

5. Default page width is 78 and default page length is 60. They may be changed by calling PGOPT (page 1599) in advance.

Description

Routine PLOTP produces a line printer plot of up to ten sets of points superimposed upon the same plot. A character "M" is printed to indicate multiple points. The user may specify the *x* and *y*-axis plot ranges and plotting symbols. Plot width and length may be reset in advance by calling PGOPT (page 1599).

PRIME

Decomposes an integer into its prime factors.

Required Arguments

N — Integer to be decomposed. (Input)

NPF — Number of different prime factors of ABS(N). (Output)
If N is equal to –1, 0, or 1, NPF is set to 0.

IPF — Integer vector of length 13. (Output)
IPF(I) contains the prime factors of the absolute value of N, for I = 1, …, NPF. The remaining 13 – NPF locations are not used.

IEXP — Integer vector of length 13. (Output)
IEXP(I) is the exponent of IPF(I), for I = 1, …, NPF. The remaining 13 – NPF locations are not used.

IPW — Integer vector of length 13. (Output)
IPW(I) contains the quantity IPF(I)**IEXP(I), for I = 1, …, NPF. The remaining 13 – NPF locations are not used.

FORTRAN 90 Interface

Generic: `CALL PRIME (N, NPF, IPF, IPW)`

Specific: The specific interface name is PRIME.

FORTRAN 77 Interface

Single: `CALL PRIME (N, NPF, IPF, IEXP, IPW)`

Example

This example factors the integer 144 = $2^4 3^2$.

```
      USE PRIME_INT
      USE UMACH_INT
      INTEGER    N
      PARAMETER  (N=144)
!
      INTEGER    IEXP(13), IPF(13), IPW(13), NOUT, NPF
!                                  Get prime factors of 144
      CALL PRIME (N, NPF, IPF, IEXP, IPW)
!                                  Get output unit number
      CALL UMACH (2, NOUT)
!                                  Print results
      WRITE (NOUT,99999) N, IPF(1), IPF(2), IEXP(1), IEXP(2), IPW(1), &
                       IPW(2), NPF
!
99999 FORMAT ('  The prime factors for', I5, ' are: ', /, 10X, 2I6, // &
            , '  IEXP =', 2I6, /, '  IPW  =', 2I6, /, '  NPF  =', I6, &
            /)
      END
```

Output

```
The prime factors for  144 are:
2     3

IEXP =     4     2
IPW  =    16     9
NPF  =     2
```

Comments

The output from `PRIME` should be interpreted in the following way: `ABS`(`N`) = `IPF`(1)`**IEXP`(1) * …. * `IPF`(`NPF`)`**IEXP`(`NPF`).

Description

Routine `PRIME` decomposes an integer into its prime factors. The number to be factored, *N*, may not have more than 13 distinct factors. The smallest number with more than 13 factors is about 1.3×10^{16}. Most computers do not allow integers of this size.

The routine `PRIME` is based on a routine by Brenner (1973).

CONST

This function returns the value of various mathematical and physical constants.

Function Return Value

CONST — Value of the constant.   (Output)
See Comment 1.

Required Arguments

NAME — Character string containing the name of the desired constant. (Input)
See Comment 3 for a list of valid constants.

FORTRAN 90 Interface

Generic: `CONST(NAME)`

Specific: The specific interface names are `S_CONST` and `D_CONST`.

FORTRAN 77 Interface

Single: `CONST(NAME)`

Double: The double precision name is `DCONST`.

Example

In this example, Euler’s constant γ is obtained and printed. Euler’s constant is defined to be

$$\gamma = \lim_{n\to\infty}\left[\sum_{k=1}^{n-1}\frac{1}{k} - \ln n\right]$$

```
      USE CONST_INT
      USE UMACH_INT
      INTEGER    NOUT
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Get gamma
      GAMA = CONST(’GAMMA’)
!                                 Print gamma
      WRITE (NOUT,*) ’GAMMA = ’, GAMA
      END
```

Output

```
GAMMA =    0.577216
```

For another example, see `CUNIT`, page 1672.

Comments

2. If the generic version of this function is used, the immediate result must be stored in a variable before use in an expression. For example:

   ```
   X = CONST(‘PI’)
   Y = COS(x)
   ```

   must be used rather than

   ```
   Y = COS(CONST(‘PI’)).
   ```

If this is too much of a restriction on the programmer, then the specific name can be used without this restriction.

2. The case of the character string in `NAME` does not matter. The names “`PI`”, “`Pi`”, “`Pi`”, and “`pi`” are equivalent.

3. The units of the physical constants are in `SI` units (meter kilogram-second).

4. The names allowed are as follows:

Name	Description	Value	Ref.
`AMU`	Atomic mass unit	1.6605402E – 27 kg	[1]
`ATM`	Standard atm pressure	1.01325E + 5N/m^2E	[2]
`AU`	Astronomical unit	1.496E + 11m	[ ]
Avogadro	Avogadro's number	6.0221367E + 231/mole	[1]
Boltzman	Boltzman's constant	1.380658E – 23J/K	[1]
`C`	Speed of light	2.997924580E + 8m/sE	[1]
Catalan	Catalan's constant	0.915965 … E	[3]
E	Base of natural logs	2.718…E	[3]
ElectronCharge	Electron change	1.60217733E –19`C`	[1]
ElectronMass	Electron mass	9.1093897E – 31 kg	[1]
ElectronVolt	Electron volt	1.60217733E – 19J	[1]
Euler	Euler's constant gamma	0.577 … E	[3]
Faraday	Faraday constant	9.6485309E + 4C/mole	[1]
FineStructure	fine structure	7.29735308E – 3	[1]
Gamma	Euler's constant	0.577 … E	[3]
Gas	Gas constant	8.314510J/mole/`k`	[1]
Gravity	Gravitational constant	6.67259E – 11N * m^2/kg^2	[1]
Hbar	Planck constant / 2 pi	1.05457266E – 34J * s	[1]
PerfectGasVolume	Std vol ideal gas	2.241383E – 2m^3/mole	[*]
Pi	Pi	3.141 … E	[3]
Planck	Planck's constant h	6.6260755E – 34J * s	[1]
ProtonMass	Proton mass	1.6726231E – 27 kg	[1]
Rydberg	Rydberg's constant	1.0973731534E + 7/m	[1]
SpeedLight	Speed of light	2.997924580E + 8m/s E	[1]
StandardGravity	Standard g	9.80665m/s^2E	[2]
StandardPressure	Standard atm pressure	1.01325E + 5N/m^2E	[2]

Name	Description	Value	Ref.
StefanBoltzmann	Stefan-Boltzman	$5.67051\text{E} - 8\text{W}/\text{K}^4/m^2$	[1]
WaterTriple	Triple point of water	$2.7316\text{E} + 2\text{K}$ E	[2]

Description

Routine CONST returns the value of various mathematical and physical quantities. For all of the physical values, the Systeme International d'Unites (SI) are used.

The reference for constants are indicated by the code in [ ] Comment above.

[1] Cohen and Taylor (1986)

[2] Liepman (1964)

[3] Precomputed mathematical constants

The constants marked with an E before the [ ] are exact (to machine precision).

To change the units of the values returned by CONST, see CUNIT, page 1672.

CUNIT

Converts X in units XUNITS to Y in units YUNITS.

Required Arguments

X — Value to be converted. (Input)

XUNITS — Character string containing the name of the units for X. (Input)
See comments for a description of units allowed.

Y — Value in YUNITS corresponding to X in XUNITS. (Output)

YUNITS — Character string containing the name of the units for Y. (Input)
See comments for a description of units allowed.

FORTRAN 90 Interface

Generic: CALL CUNIT (X, XUNITS, Y, YUNITS [,…])

Specific: The specific interface names are S_CUNIT and D_CUNIT.

FORTRAN 77 Interface

Single: CALL CUNIT (X, XUNITS, Y, YUNITS)

Double: The double precision name is DCUNIT.

Example

The routine CONST is used to obtain the speed on light, *c*, in SI units. CUNIT is then used to convert *c* to mile/second and to parsec/year. An example involving substitution of force for mass is required in conversion of Newtons/Meter^2 to Pound/Inch^2.

```
      USE CONST_INT
      USE CUNIT_INT
      USE UMACH_INT
!      INTEGER    NOUT
      REAL       CMH, CMS, CPY
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Get speed of light in SI (m/s)
      CMS = CONST(’SpeedLight’)
      WRITE (NOUT,*) ’Speed of Light = ’, CMS, ’ meter/second’
!                                 Get speed of light in mile/second
      CALL CUNIT (CMS, ’SI’, CMH, ’Mile/Second’)
      WRITE (NOUT,*) ’Speed of Light = ’, CMH, ’ mile/second’
!                                 Get speed of light in parsec/year
      CALL CUNIT (CMS, ’SI’, CPY, ’Parsec/Year’)
      WRITE (NOUT,*) ’Speed of Light = ’, CPY, ’ Parsec/Year’
!                                 Convert Newton/Meter**2 to
!                                 Pound/Inch**2.
      CALL CUNIT(1.E0, ’Newton/Meter**2’, CPSI, &
                      ’Pound/Inch**2’)
      WRITE(NOUT,*)’ Atmospheres, in Pound/Inch**2 = ’,CPSI
      END
```

Output

```
Speed of Light =     2.99792E+08 meter/second
Speed of Light =     186282. mile/second
Speed of Light =    0.306387 Parsec/Year

*** WARNING  ERROR 8 from CUNIT.  A conversion of units of mass to units of
***          force was required for consistency.
Atmospheres, in Pound/Inch**2 =     1.45038E-04
```

Comments

1. Strings XUNITS and YUNITS have the form $U_1 * U_2 * \dots * U_m/V_1 \dots V_n$, where U_i and V_i are the names of basic units or are the names of basic units raised to a power. Examples are, “METER * KILOGRAM/SECOND”, “M * KG/S”, “METER”, or “M/KG^2”.

2. The case of the character string in XUNITS and YUNITS does not matter. The names “METER”, “Meter” and “meter” are equivalent.

3. If XUNITS is “SI”, then X is assumed to be in the standard international units corresponding to YUNITS. Similarly, if YUNITS is

   “SI”, then Y is assumed to be in the standard international units corresponding to XUNITS.

4. The basic unit names allowed are as follows:

Units of time
day, hour = hr, min = minute, s = sec = second, year

Units of frequency
Hertz = Hz

Units of mass
AMU, g = gram, lb = pound, ounce = oz, slug

Units of distance
Angstrom, AU, feet = foot = ft, in = inch, m = meter = metre, micron, mile, mill, parsec, yard

Units of area
acre

Units of volume
l = liter = litre

Units of force
dyne, N = Newton, poundal

Units of energy
BTU(thermochemical), Erg, J = Joule

Units of work
W = watt

Units of pressure
ATM = atomosphere, bar, Pascal

Units of temperature
degC = Celsius, degF = Fahrenheit, degK = Kelvin

Units of viscosity
poise, stoke

Units of charge
Abcoulomb, C = Coulomb, statcoulomb

Units of current
A = ampere, abampere, statampere,

Units of voltage
Abvolt, V = volt

Units of magnetic induction
T = Tesla, Wb = Weber

Other units
1, farad, mole, Gauss, Henry, Maxwell, Ohm

The following metric prefixes may be used with the above units. Note that the one or two letter prefixes may only be used with one letter unit abbreviations.

A	Atto	1.E – 18
F	femto	1.E – 15
P	Pico	1.E – 12
N	nano	1.E – 9
U	micro	1.E – 6
M	milli	1.E – 3
C	centi	1.E – 2
D	Deci	1.E – 1
DK	Deca	1.E + 2
K	Kilo	1.E + 3
	myria	1.E + 4 (no single letter prefix; M means milli
	mega	1.E + 6 (no single letter prefix; M means milli
G	Giga	1.E + 9
T	Tera	1.E + 12

5. Informational error

Type	Code	
3	8	A conversion of units of mass to units of force was required for consistency.

Description

Routine CUNIT converts a value expressed in one set of units to a value expressed in another set of units.

The input and output units are checked for consistency unless the input unit is “SI”. SI means the Systeme International d’Unites. This is the meter–kilogram–second form of the metric system. If the input units are “SI”, then the input is assumed to be expressed in the SI units consistent with the output units.

HYPOT

This functions computes SQRT(A**2 + B**2) without underflow or overflow.

Function Return Value

HYPOT — SQRT(A**2 + B**2). (Output)

Required Arguments

A — First parameter. (Input)

B — Second parameter. (Input)

FORTRAN 90 Interface

Generic: `HYPOT(A, B)`

Specific: The specific interface names are `S_HYPOT` and `D_HYPOT`.

FORTRAN 77 Interface

Single: `HYPOT(A, B)`

Double: The double precision name is `DHYPOT`.

Example

Computes

$$c = \sqrt{a^2 + b^2}$$

where $a = 10^{20}$ and $b = 2 \times 10^{20}$ without overflow.

```
      USE HYPOT_INT
      USE UMACH_INT
!                                 Declare variables
      INTEGER    NOUT
      REAL       A, B, C
!
      A = 1.0E+20
      B = 2.0E+20
      C = HYPOT(A,B)
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Print the results
      WRITE (NOUT,'(A,1PE10.4)') ' C = ', C
      END
```

Output

```
C = 2.2361E+20
```

Description

Routine `HYPOT` is based on the routine `PYTHAG`, used in EISPACK 3. This is an update of the work documented in Garbow et al. (1972).

Reference Material

Contents

User Errors

IMSL routines attempt to detect user errors and handle them in a way that provides as much information to the user as possible. To do this, we recognize various levels of severity of errors, and we also consider the extent of the error in the context of the purpose of the routine; a trivial error in one situation may be serious in another. IMSL routines attempt to report as many errors as they can reasonably detect. Multiple errors present a difficult problem in error detection because input is interpreted in an uncertain context after the first error is detected.

What Determines Error Severity

In some cases, the user's input may be mathematically correct, but because of limitations of the computer arithmetic and of the algorithm used, it is not possible to compute an answer accurately. In this case, the assessed degree of accuracy determines the severity of the error. In cases where the routine computes several output quantities, if some are not computable but most are, an error condition exists. The severity depends on an assessment of the overall impact of the error.

Terminal errors

If the user's input is regarded as meaningless, such as N = −1 when "N" is the number of equations, the routine prints a message giving the value of the erroneous input argument(s) and the reason for the erroneous input. The routine will then cause the user's program to stop. An error in which the user's input is meaningless is the most severe error and is called a *terminal error*. Multiple terminal error messages may be printed from a single routine.

Informational errors

In many cases, the best way to respond to an error condition is simply to correct the input and rerun the program. In other cases, the user may want to take actions in the program itself based on errors that occur. An error that may be used as the basis for corrective action within the program is called an *informational error*. If an informational error occurs, a user-retrievable code is set. A routine can return at most one informational error for a single reference to the routine. The codes for the informational error codes are printed in the error messages.

Other errors

In addition to informational errors, IMSL routines issue error messages for which no user-retrievable code is set. Multiple error messages for this kind of error may be printed. These errors, which generally are not described in the documentation, include terminal errors as well as less serious errors. Corrective action within the calling program is not possible for these errors.

Kinds of Errors and Default Actions

Five levels of severity of errors are defined in the MATH/LIBRARY. Each level has an associated PRINT attribute and a STOP attribute. These attributes have default settings (YES or NO), but they may also be set by the user. The purpose of having multiple error severity levels is to provide independent control of actions to be taken for errors of different severity. Upon return from an IMSL routine, exactly one error state exists. (A code 0 “error” is no informational error.) Even if more than one informational error occurs, only one message is printed (if the PRINT attribute is YES). Multiple errors for which no corrective action within the calling program is reasonable or necessary result in the printing of multiple messages (if the PRINT attribute for their severity level is YES). Errors of any of the severity levels except level 5 may be informational errors.

Level 1: **Note**. A *note* is issued to indicate the possibility of a trivial error or simply to provide information about the computations. Default attributes: PRINT=NO, STOP=NO

Level 2: **Alert**. An *alert* indicates that the user should be advised about events occurring in the software. Default attributes: PRINT=NO, STOP=NO

Level 3: **Warning**. A *warning* indicates the existence of a condition that may require corrective action by the user or calling routine. A warning error may be issued because the results are accurate to only a few decimal places, because some of the output may be erroneous but most of the output is correct, or because some assumptions underlying the analysis technique are violated. Often no corrective action is necessary and the condition can be ignored. Default attributes: PRINT=YES, STOP=NO

Level 4: **Fatal**.A *fatal* error indicates the existence of a condition that may be serious. In most cases, the user or calling routine must take corrective action to recover. Default attributes: PRINT=YES, STOP=YES

Level 5: **Terminal**.A *terminal* error is serious. It usually is the result of an incorrect specification, such as specifying a negative number as the number of equations. These errors may also be caused by various programming errors impossible to diagnose correctly in FORTRAN. The resulting error message may be perplexing to the user. In such cases, the user is advised to compare carefully the actual arguments passed to the routine with the dummy argument descriptions given in the documentation. Special attention should be given to checking argument order and data types.

A terminal error is not an informational error because corrective action within the program is generally not reasonable. In normal usage, execution is terminated immediately when a terminal error occurs. Messages relating to more than one terminal error are printed if they occur. Default attributes: PRINT=YES, STOP=YES

The user can set PRINT and STOP attributes by calling `ERSET` as described in “Routines for Error Handling.”

Errors in Lower-Level Routines

It is possible that a user’s program may call an IMSL routine that in turn calls a nested sequence of lower-level IMSL routines. If an error occurs at a lower level in such a nest of routines and if the lower-level routine cannot pass the information up to the original user-called routine, then a traceback of the routines is produced. The only common situation in which this can occur is when an IMSL routine calls a user-supplied routine that in turn calls another IMSL routine.

Routines for Error Handling

There are three ways in which the user may interact with the IMSL error handling system: (1) to change the default actions, (2) to retrieve the integer code of an informational error so as to take corrective action, and (3) to determine the severity level of an error. The routines to use are `ERSET`, `IERCD`, and `N1RTY`, respectively.

ERSET

Change the default printing or stopping actions when errors of a particular error severity level occur.

Required Arguments

IERSVR — Error severity level indicator. (Input)
If `IERSVR` = 0, actions are set for levels 1 to 5. If `IERSVR` is 1 to 5, actions are set for errors of the specified severity level.

IPACT — Printing action. (Input)

IPACT	**Action**
−1	Do not change current setting(s).
0	Do not print.
1	Print.
2	Restore the default setting(s).

ISACT — Stopping action. (Input)

ISACT	**Action**
−1	Do not change current setting(s).
0	Do not stop.
1	Stop.
2	Restore the default setting(s).

FORTRAN 90 Interface

Generic: `CALL ERSET (IERSVR, IPACT, ISACT)`

Specific: The specific interface name is `ERSET`.

FORTRAN 77 Interface

Single: `CALL ERSET (IERSVR, IPACT, ISACT)`

IERCD and N1RTY

The last two routines for interacting with the error handling system, `IERCD` and `N1RTY`, are `INTEGER` functions and are described in the following material.

`IERCD` retrieves the integer code for an informational error. Since it has no arguments, it may be used in the following way:

`ICODE = IERCD( )`

The function retrieves the code set by the most recently called IMSL routine.

`N1RTY` retrieves the error type set by the most recently called IMSL routine. It is used in the following way:

`ITYPE = N1RTY(1)`

`ITYPE` = 1, 2, 4, and 5 correspond to error severity levels 1, 2, 4, and 5, respectively. `ITYPE` = 3 and `ITYPE` = 6 are both warning errors, error severity level 3. While `ITYPE` = 3 errors are informational errors (`IERCD`( ) ≠ 0), `ITYPE` = 6 errors are not informational errors (`IERCD`( ) = 0).

For software developers requiring additional interaction with the IMSL error handling system, see Aird and Howell (1991).

Examples

Changes to default actions

Some possible changes to the default actions are illustrated below. The default actions remain in effect for the kinds of errors not included in the call to `ERSET`.

To turn off printing of warning error messages:

```
CALL ERSET (3, 0, −1)
```

To stop if warning errors occur:

```
CALL ERSET (3, −1, 1)
```

To print all error messages:

```
CALL ERSET (0, 1, −1)
```

To restore all default settings:

```
CALL ERSET (0, 2, 2)
```

Use of informational error to determine program action

In the program segment below, the Cholesky factorization of a matrix is to be performed. If it is determined that the matrix is not nonnegative definite (and often this is not immediately obvious), the program is to take a different branch.

```
                     .
                     .
                     .
      CALL LFTDS (A, FACT)
      IF (IERCD() .EQ. 2) THEN
!                     Handle matrix that is not nonnegative definite
                     .
                     .
                     .
      END IF
```

Examples of errors

The program below illustrates each of the different types of errors detected by the MATH/LIBRARY routines.

The error messages refer to the argument names that are used in the documentation for the routine, rather than the user’s name of the variable used for the argument. In the message generated by IMSL routine LINRG in this example, reference is made to N, whereas in the program a literal was used for this argument.

```
      USE_IMSL_LIBRARIES
      INTEGER    N
      PARAMETER  (N=2)
!
      REAL       A(N,N), AINV(N,N), B(N), X(N)
!
      DATA A/2.0, -3.0, 2.0, -3.0/
      DATA B/1.0, 2.0/
!                                  Turn on printing and turn off
!                                  stopping for all error types.
      CALL ERSET (0, 1, 0)
!                                  Generate level 4 informational error.
      CALL LSARG (A, B, X)
!                                  Generate level 5 terminal error.
      CALL LINRG (A, AINV, N = -1)
      END
```

Output

```
*** FATAL    ERROR 2 from LSARG.  The input matrix is singular.  Some of
***          the diagonal elements of the upper triangular matrix U of the
***          LU factorization are close to zero.

*** TERMINAL ERROR 1 from LINRG.  The order of the matrix must be positive
***          while N = –1 is given.
```

Example of traceback

The next program illustrates a situation in which a traceback is produced. The program uses the IMSL quadrature routines QDAG and QDAGS to evaluate the double integral

$$\int_0^1 \int_0^1 (x+y)\,dx\,dy = \int_0^1 g(y)\,dy$$

where

$$g(y) = \int_0^1 (x+y)\,dx = \int_0^1 f(x)\,dx, \text{ with } f(x) = x+y$$

Since both QDAG and QDAGS need 2500 numeric storage units of workspace, and since the workspace allocator uses some space to keep track of the allocations, 6000 numeric storage units of space are explicitly allocated for workspace. Although the traceback shows an error code associated with a terminal error, this code has no meaning to the user; the printed message contains all relevant information. It is not assumed that the user would take corrective action based on knowledge of the code.

```
      USE QDAGS_INT
!                                 Specifications for local variables
      REAL       A, B, ERRABS, ERREST, ERRREL, G, RESULT
      EXTERNAL   G
!                                 Set quadrature parameters
      A      = 0.0
      B      = 1.0
      ERRABS = 0.0
      ERRREL = 0.001
!                                 Do the outer integral
      CALL QDAGS (G, A, B, RESULT, ERRABS, ERRREL, ERREST)
!
      WRITE (*,*) RESULT, ERREST
      END
!
      REAL FUNCTION G (ARGY)
      USE QDAG_INT
      REAL       ARGY
!
      INTEGER    IRULE
      REAL       C, D, ERRABS, ERREST, ERRREL, F, Y
      COMMON     /COMY/ Y
      EXTERNAL   F
!
      Y      = ARGY
      C      = 0.0
      D      = 1.0
      ERRABS = 0.0
      ERRREL = -0.001
      IRULE  = 1
!
      CALL QDAG (F, C, D, G, ERRABS, ERRREL, IRULE, ERREST)
      RETURN
      END
!
      REAL FUNCTION F (X)
      REAL       X
!
      REAL       Y
      COMMON     /COMY/ Y
!
      F = X + Y
      RETURN
      END
```

Output

```
*** TERMINAL ERROR 4 from Q2AG.  The relative error desired ERRREL =
***          -1.000000E-03.  It must be at least zero.
Here is a traceback of subprogram calls in reverse order:
Routine name                    Error type  Error code
------------                    ----------  ----------
Q2AG                                5           4    (Called internally)
QDAG                                0           0
Q2AGS                               0           0    (Called internally)
QDAGS                               0           0
USER                                0           0
```

Machine-Dependent Constants

The function subprograms in this section return machine-dependent information and can be used to enhance portability of programs between different computers. The routines `IMACH`, and `AMACH` describe the computer's arithmetic. The routine `UMACH` describes the input, ouput, and error output unit numbers.

IMACH

This function retrieves machine integer constants that define the arithmetic used by the computer.

Function Return Value

`IMACH`(1) = Number of bits per integer storage unit.

`IMACH`(2) = Number of characters per integer storage unit:

Integers are represented in *M*-digit, base *A* form as

$$\sigma\sum_{k=0}^{M} x_k A^k$$

where σ is the sign and $0 \le x_k < A$, $k = 0, \ldots, M$.

Then,

`IMACH`(3) = A, the base.

`IMACH`(4) = M, the number of base-A digits.

`IMACH`(5) = $A^M - 1$, the largest integer.

The machine model assumes that floating-point numbers are represented in normalized N-digit, base B form as

$$\sigma B^E \sum_{k=1}^{N} x_k B^{-k}$$

where σ is the sign, $0 < x_1 < B$, $0 \le x_k < B$, k = 2, …, N and $E_{\min} \le E \le E_{\max}$. Then,

IMACH(6) = B, the base.

IMACH(7) = N_s, the number of base-B digits in single precision.

IMACH(8) = $E_{\min_s}$, the smallest single precision exponent.

IMACH(9) = $E_{\max_s}$, the largest single precision exponent.

IMACH(10) = N_d, the number of base-B digits in double precision.

IMACH(11) = $E_{\min_d}$, the smallest double precision exponent.

IMACH(12) = $E_{\max_d}$, the number of base-B digits in double precision

Required Arguments

I — Index of the desired constant. (Input)

FORTRAN 90 Interface

Generic: `IMACH (I)`

Specific: The specific interface name is `IMACH`.

FORTRAN 77 Interface

Single: `IMACH (I)`

AMACH

The function subprogram `AMACH` retrieves machine constants that define the computer's single-precision or double precision arithmetic. Such floating-point numbers are represented in normalized *N*-digit, base *B* form as

$$\sigma B^E \sum_{k=1}^{N} x_k B^{-k}$$

where σ is the sign, $0 < x_1 < B$, $0 \le x_k < B$, $k = 2, \ldots, N$ and

$$E_{\min} \le E \le E_{\max}$$

Function Return Value

AMACH(1) = $B^{E_{\min}-1}$, the smallest normalized positive number.

AMACH(2)=$B^{E_{\max}}\left(1-B^{-N}\right)$, the largest number.

AMACH(3)=B^{-N}, the smallest relative spacing.

AMACH(4)=B^{1-N}, the largest relative spacing.

AMACH(5) = $\log_{10}\left(B\right)$.

AMACH(6) = NaN (*quiet* not a number).

AMACH(7)=positive machine infinity.

AMACH(8)= negative machine infinity.

See Comment 1 for a description of the use of the generic version of this function.

See Comment 2 for a description of `min`, `max`, and `N`.

Required Arguments

I — Index of the desired constant. (Input)

FORTRAN 90 Interface

Generic: `AMACH (I)`

Specific: The specific interface names are `S_AMACH` and `D_AMACH`.

FORTRAN 77 Interface

Single: `AMACH (I)`

Double: The double precision name is `DMACH`.

Comments

1. If the generic version of this function is used, the immediate result must be stored in a variable before use in an expression. For example:

   ```
   X = AMACH(I)
   Y = SQRT(X)
   ```

   must be used rather than

   ```
   Y = SQRT(AMACH(I)).
   ```

   If this is too much of a restriction on the programmer, then the specific name can be used without this restriction.

2. Note that for single precision *B* = `IMACH`(6), *N* = `IMACH`(7).
   E`min` = `IMACH`(8), and *E*`max`, = `IMACH`(9).
   For double precision *B* = `IMACH`(6), *N* = `IMACH`(10).
   E`min` = `IMACH`(11), and *E*`max`, = `IMACH`(12).

3. The IEEE standard for binary arithmetic (see IEEE 1985) specifies *quiet* NaN (not a number) as the result of various invalid or ambiguous operations, such as 0/0. The intent is that `AMACH`(6) return a *quiet* NaN. On IEEE format computers that do not support a quiet NaN, a special value near `AMACH`(2) is returned for `AMACH`(6). On computers that do not have a special representation for infinity, `AMACH`(7) returns the same value as `AMACH`(2).

DMACH

See `AMACH`.

IFNAN(X)

This logical function checks if the argument `X` is NaN (not a number).

Function Return Value

***IFNAN* -** Logical function value. True is returned if the input argument is a `NAN`**.** Otherwise, False is returned. (Output)

Required Arguments

X – Argument for which the test for `NAN` is desired. (Input)

FORTRAN 90 Interface

Generic: `IFNAN(X)`

Specific: The specific interface names are S_IFNAN and D_IFNAN.

FORTRAN 77 Interface

Single: IFNAN (X)

Double: The double precision name is DIFNAN.

Example

```
      USE IFNAN_INT
      USE AMACH_INT
      USE UMACH_INT
      INTEGER      NOUT
      REAL         X
!
      CALL UMACH (2, NOUT)
!
      X = AMACH(6)
      IF (IFNAN(X)) THEN
         WRITE (NOUT,*) ' X is NaN (not a number).'
      ELSE
         WRITE (NOUT,*) ' X = ', X
      END IF
!
      END
```

Output

```
X is NaN (not a number).
```

Description

The logical function IFNAN checks if the single or double precision argument X is NAN (not a number). The function IFNAN is provided to facilitate the transfer of programs across computer systems. This is because the check for NaN can be tricky and not portable across computer systems that do not adhere to the IEEE standard. For example, on computers that support the IEEE standard for binary arithmetic (see IEEE 1985), NaN is specified as a bit format not equal to itself. Thus, the check is performed as

```
IFNAN = X .NE. X
```

On other computers that do not use IEEE floating-point format, the check can be performed as:

```
IFNAN = X .EQ. AMACH(6)
```

The function IFNAN is equivalent to the specification of the function Isnan listed in the Appendix, (IEEE 1985). The above following example illustrates the use of IFNAN. If X is NaN, a message is printed instead of X. (Routine UMACH, which is described in the following section, is used to retrieve the output unit number for printing the message.)

UMACH

Routine UMACH sets or retrieves the input, output, or error output device unit numbers.

Required Arguments

N — Integer value indicating the action desired. If the value of N is negative, the input, output, or error output unit number is reset to NUNIT. If the value of N is positive, the input, output, or error output unit number is returned in NUNIT. See the table in argument NUNIT for legal values of N. (Input)

NUNIT — The unit number that is either retrieved or set, depending on the value of input argument N. (Input/Output)

The arguments are summarized by the following table:

N	Effect
1	Retrieves input unit number in NUNIT.
2	Retrieves output unit number in NUNIT.
3	Retrieves error output unit number in NUNIT.
−1	Sets the input unit number to NUNIT.
−2	Sets the output unit number to NUNIT.
−3	Sets the error output unit number to NUNIT.

FORTRAN 90 Interface

Generic: CALL UMACH (N, NUNIT)

Specific: The specific interface name is UMACH.

FORTRAN 77 Interface

Single: CALL UMACH (N, NUNIT)

Example

In the following example, a terminal error is issued from the MATH/LIBRARY AMACH function since the argument is invalid. With a call to UMACH, the error message will be written to a local file named “CHECKERR”.

```
      USE AMACH_INT
      USE UMACH_INT
      INTEGER      N, NUNIT
      REAL         X
!                                  Set Parameter
```

```
      N = 0
      NUNIT = 9
!
      CALL UMACH (-3, NUNIT)
      OPEN (UNIT=NUNIT,FILE='CHECKERR')
      X = AMACH(N)
      END
```

Output

The output from this example, written to “CHECKERR” is:

```
*** TERMINAL ERROR 5 from AMACH.  The argument must be between 1 and 8
***           inclusive. N = 0
```

Description

Routine UMACH sets or retrieves the input, output, or error output device unit numbers. UMACH is set automatically so that the default FORTRAN unit numbers for standard input, standard output, and standard error are used. These unit numbers can be changed by inserting a call to UMACH at the beginning of the main program that calls MATH/LIBRARY routines. If these unit numbers are changed from the standard values, the user should insert an appropriate OPEN statement in the calling program.

Matrix Storage Modes

In this section, the word *matrix* will be used to refer to a mathematical object, and the word *array* will be used to refer to its representation as a FORTRAN data structure.

General Mode

A *general* matrix is an $N \times N$ matrix A. It is stored in a FORTRAN array that is declared by the following statement:

```
DIMENSION A(LDA,N)
```

The parameter LDA is called the *leading dimension* of A. It must be at least as large as N. IMSL general matrix subprograms only refer to values A_{ij} for $i = 1, \ldots, N$ and $j = 1, \ldots, N$. The data type of a general array can be one of REAL, DOUBLE PRECISION, or COMPLEX. If your FORTRAN compiler allows, the nonstandard data type DOUBLE COMPLEX can also be declared.

Rectangular Mode

A *rectangular* matrix is an $M \times N$ matrix A. It is stored in a FORTRAN array that is declared by the following statement:

```
DIMENSION A(LDA,N)
```

The parameter LDA is called the *leading dimension* of A. It must be at least as large as M. IMSL rectangular matrix subprograms only refer to values A_{ij} for $i = 1, \ldots, M$ and $j = 1, \ldots, N$. The data type of a rectangular array can be REAL, DOUBLE PRECISION, or COMPLEX. If your FORTRAN compiler allows, you can declare the nonstandard data type DOUBLE COMPLEX.

Symmetric Mode

A symmetric matrix is a square $N \times N$ matrix A, such that $A^T = A$. (A^T is the transpose of A.) It is stored in a FORTRAN array that is declared by the following statement:

```
DIMENSION A(LDA,N)
```

The parameter `LDA` is called the *leading dimension* of `A`. It must be at least as large as `N`. IMSL symmetric matrix subprograms only refer to the upper or to the lower half of `A` (i.e., to values A_{ij} for $i = 1, \ldots, N$ and $j = i, \ldots, N$, or A_{ij} for $j = 1, \ldots, N$ and $i = j, \ldots, N$). The data type of a symmetric array can be one of `REAL` or `DOUBLE PRECISION`. Use of the upper half of the array is denoted in the BLAS that compute with symmetric matrices, see Chapter 9, Programming Notes for BLAS, using the `CHARACTER*1` flag `UPLO = 'U'`. Otherwise, `UPLO = 'L'` denotes that the lower half of the array is used.

Hermitian Mode

A *Hermitian* matrix is a square $N \times N$ matrix A, such that

$$\overline{A}^T = A$$

The matrix

$$\overline{A}$$

is the complex conjugate of A and

$$A^H \equiv \overline{A}^T$$

is the conjugate transpose of A. For Hermitian matrices, $A^H = A$. The matrix is stored in a FORTRAN array that is declared by the following statement:

```
DIMENSION A(LDA,N)
```

The parameter `LDA` is called the *leading dimension* of `A`. It must be at least as large as N. IMSL Hermitian matrix subprograms only refer to the upper or to the lower half of `A` (i.e., to values A_{ij} for $i = 1, \ldots, N$ and $j = i, \ldots, N$., or A_{ij} for $j = 1, \ldots, N$ and $i = j, \ldots, N$). Use of the upper half of the array is denoted in the BLAS that compute with Hermitian matrices, see Chapter 9, Programming Notes for BLAS, using the `CHARACTER*1` flag `UPLO = 'U'`. Otherwise, `UPLO = 'L'` denotes that the lower half of the array is used. The data type of a Hermitian array can be `COMPLEX` or, if your FORTRAN compiler allows, the nonstandard data type `DOUBLE COMPLEX`.

Triangular Mode

A *triangular* matrix is a square $N \times N$ matrix A such that values $A_{ij} = 0$ for $i < j$ or $A_{ij} = 0$ for $i > j$. The first condition defines a *lower* triangular matrix while the second condition defines an *upper* triangular matrix. A lower triangular matrix A is stored in the lower triangular part of a FORTRAN array `A`. An upper triangular matrix is stored in the upper triangular part of a FORTRAN array. Triangular matrices are called *unit* triangular whenever $A_{jj} = 1, j = 1, \ldots, N$. For unit triangular matrices, only the strictly lower or upper parts of the array are referenced. This is denoted in the BLAS that compute with triangular matrices, see Chapter 9, Programming Notes for BLAS, using the `CHARACTER*1` flag `DIAGNL = 'U'`. Otherwise, `DIAGNL = 'N'` denotes that the diagonal array terms should be used. For unit triangular matrices, the diagonal terms are each used with the mathematical value 1. The array diagonal term does not need to be 1.0 in this usage. Use of the upper half of the array is denoted in the BLAS that compute with triangular matrices, see Chapter 9, Programming Notes for BLAS, using the `CHARACTER*1` flag `UPLO = 'U'`. Otherwise, `UPLO = 'L'` denotes that the lower half of the array is used. The data type of an array that contains a triangular matrix can be one of `REAL`, `DOUBLE PRECISION`, or `COMPLEX`. If your FORTRAN compiler allows, the nonstandard data type `DOUBLE COMPLEX` can also be declared.

Band Storage Mode

A *band matrix* is an $M \times N$ matrix A with all of its nonzero elements "close" to the main diagonal. Specifically, values $A_{ij} = 0$ if $i - j >$ `NLCA` or $j - i >$ `NUCA`. The integers `NLCA` and `NUCA` are the *lower* and *upper* band widths. The integer $m =$ `NLCA` + `NUCA` + 1 is the total band width. The diagonals, other than the main diagonal, are called *codiagonals*. While any $M \times N$ matrix is a band matrix, the band matrix mode is most useful only when the number of nonzero codiagonals is much less than m.

In the band storage mode, the `NLCA` lower codiagonals and `NUCA` upper codiagonals are stored in the rows of a FORTRAN array of dimension $m \times N$. The elements are stored in the same column of the array as they are in the matrix. The values A_{ij} inside the band width are stored in array positions ($i - j$ + `NUCA` + 1, j). This array is declared by the following statement:

```
DIMENSION A(LDA,N)
```

The parameter `LDA` is called the *leading dimension* of A. It must be at least as large as m. The data type of a band matrix array can be one of `REAL`, `DOUBLE PRECISION`, `COMPLEX` or, if your FORTRAN compiler allows, the nonstandard data type `DOUBLE COMPLEX` . Use of the `CHARACTER*1` flag `TRANS='N'` in the BLAS, , see Chapter 9, Programming Notes for BLAS, specifies that the matrix A is used. The flag value

$$\text{TRANS ='T' uses } A^T$$

while

$$\text{TRANS ='C' uses } \overline{A}^T$$

For example, consider a real 5 × 5 band matrix with 1 lower and 2 upper codiagonals, stored in the FORTRAN array declared by the following statements:

```
PARAMETER (N=5, NLCA=1, NUCA=2)
REAL A(NLCA+NUCA+1, N)
```

The matrix A has the form

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & 0 & 0 \\ A_{21} & A_{22} & A_{23} & A_{24} & 0 \\ 0 & A_{32} & A_{33} & A_{34} & A_{35} \\ 0 & 0 & A_{43} & A_{44} & A_{45} \\ 0 & 0 & 0 & A_{54} & A_{55} \end{bmatrix}$$

As a FORTRAN array, it is

$$A = \begin{bmatrix} \times & \times & A_{13} & A_{24} & A_{35} \\ \times & A_{12} & A_{23} & A_{34} & A_{45} \\ A_{11} & A_{22} & A_{33} & A_{44} & A_{55} \\ A_{21} & A_{32} & A_{43} & A_{54} & \times \end{bmatrix}$$

The entries marked with an `x` in the above array are not referenced by the IMSL band subprograms.

Band Symmetric Storage Mode

A *band symmetric* matrix is a band matrix that is also symmetric. The band symmetric storage mode is similar to the band mode except only the lower or upper codiagonals are stored.

In the band symmetric storage mode, the `NCODA` upper codiagonals are stored in the rows of a FORTRAN array of dimension (`NCODA` + `1`) × N. The elements are stored in the same column of the array as they are in the matrix. Specifically, values $A_{ij}, j \le i$ inside the band are stored in array positions ($i - j$ + `NCODA` + 1, j). This is the storage mode designated by using the `CHARACTER*`1 flag `UPLO = 'U'` in Level 2 BLAS that compute with band symmetric matrices, , see Chapter 9, Programming Notes for BLAS. Alternatively, $A_{ij}, j \le i$, inside the band, are stored in array positions $(i - j + 1, j)$. This is the storage mode designated by using the `CHARACTER*`1 flag `UPLO = 'L'` in these Level 2 BLAS, see Chapter 9, Programming Notes for BLAS. The array is declared by the following statement:

```
DIMENSION A(LDA,N)
```

The parameter `LDA` is called the *leading dimension* of A. It must be at least as large as `NCODA` + 1. The data type of a band symmetric array can be `REAL` or `DOUBLE PRECISION`.

For example, consider a real 5 × 5 band matrix with 2 codiagonals. Its FORTRAN declaration is

```
PARAMETER (N=5, NCODA=2)
REAL A(NCODA+1, N)
```

The matrix A has the form

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & 0 & 0 \\ A_{12} & A_{22} & A_{23} & A_{24} & 0 \\ A_{13} & A_{23} & A_{33} & A_{34} & A_{35} \\ 0 & A_{24} & A_{34} & A_{44} & A_{45} \\ 0 & 0 & A_{35} & A_{45} & A_{55} \end{bmatrix}$$

Since A is symmetric, the values $A_{ij} = A_{ji}$. In the FORTRAN array, it is

$$A = \begin{bmatrix} \times & \times & A_{13} & A_{24} & A_{35} \\ \times & A_{12} & A_{23} & A_{34} & A_{45} \\ A_{11} & A_{22} & A_{33} & A_{44} & A_{55} \end{bmatrix}$$

The entries marked with an × in the above array are not referenced by the IMSL band symmetric subprograms.

An alternate storage mode for band symmetric matrices is designated using the `CHARACTER*1` flag `UPLO = 'L'` in Level 2 BLAS that compute with band symmetric matrices, see Chapter 9, Programming Notes for BLAS. In that case, the example matrix is represented as

$$A = \begin{bmatrix} A_{11} & A_{22} & A_{33} & A_{44} & A_{55} \\ A_{12} & A_{23} & A_{34} & A_{45} & \times \\ A_{13} & A_{24} & A_{35} & \times & \times \end{bmatrix}$$

Band Hermitian Storage Mode

A *band Hermitian* matrix is a band matrix that is also Hermitian. The band Hermitian mode is a complex analogue of the band symmetric mode.

In the band Hermitian storage mode, the `NCODA` upper codiagonals are stored in the rows of a FORTRAN array of dimension (`NCODA` + 1) × *N*. The elements are stored in the same column of the array as they are in the matrix. In the Level 2 BLAS, see Chapter 9, Programming Notes for BLAS, this is denoted by using the `CHARACTER*1` flag `UPLO ='U'`. The array is declared by the following statement:

```
DIMENSION A(LDA,N)
```

The parameter `LDA` is called the *leading dimension* of *A*. It must be at least as large as (`NCODA` + 1) . The data type of a band Hermitian array can be `COMPLEX` or, if your FORTRAN compiler allows, the nonstandard data type `DOUBLE COMPLEX`.

For example, consider a complex 5 × 5 band matrix with 2 codiagonals. Its FORTRAN declaration is

```
PARAMETER (N=5, NCODA = 2)
COMPLEX A(NCODA + 1, N)
```

The matrix *A* has the form

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & 0 & 0 \\ \overline{A}_{12} & A_{22} & A_{23} & A_{24} & 0 \\ \overline{A}_{13} & \overline{A}_{23} & A_{33} & A_{34} & A_{35} \\ 0 & \overline{A}_{24} & \overline{A}_{34} & A_{44} & A_{45} \\ 0 & 0 & \overline{A}_{35} & \overline{A}_{45} & A_{55} \end{bmatrix}$$

where the value

$$\overline{A}_{ij}$$

is the complex conjugate of A_{ij}. This matrix represented as a FORTRAN array is

$$A = \begin{bmatrix} \times & \times & A_{13} & A_{24} & A_{35} \\ \times & A_{12} & A_{23} & A_{34} & A_{45} \\ A_{11} & A_{22} & A_{33} & A_{44} & A_{55} \end{bmatrix}$$

The entries marked with an × in the above array are not referenced by the IMSL band Hermitian subprograms.

An alternate storage mode for band Hermitian matrices is designated using the CHARACTER*1 flag UPLO = 'L' in Level 2 BLAS that compute with band Hermitian matrices, see Chapter 9, Programming Notes for BLAS. In that case, the example matrix is represented as

$$A = \begin{bmatrix} A_{11} & A_{22} & A_{33} & A_{44} & A_{55} \\ \overline{A}_{12} & \overline{A}_{23} & \overline{A}_{34} & \overline{A}_{45} & \times \\ \overline{A}_{13} & \overline{A}_{24} & \overline{A}_{35} & \times & \times \end{bmatrix}$$

Band Triangular Storage Mode

A *band triangular* matrix is a band matrix that is also triangular. In the band triangular storage mode, the NCODA codiagonals are stored in the rows of a FORTRAN array of dimension (NCODA + 1) × N. The elements are stored in the same column of the array as they are in the matrix. For usage in the Level 2 BLAS, see Chapter 9, Programming Notes for BLAS, the CHARACTER*1 flag DIAGNL has the same meaning as used in section “Triangular Storage Mode”. The flag UPLO has the meaning analogous with its usage in the section “Banded Symmetric Storage Mode”. This array is declared by the following statement:

```
DIMENSION A(LDA,N)
```

The parameter LDA is called the *leading dimension* of A. It must be at least as large as (NCODA + 1).

For example, consider a 5 ×5 band upper triangular matrix with 2 codiagonals. Its FORTRAN declaration is

```
PARAMETER (N = 5, NCODA = 2)
COMPLEX A(NCODA + 1, N)
```

The matrix A has the form

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & 0 & 0 \\ 0 & A_{22} & A_{23} & A_{24} & 0 \\ 0 & 0 & A_{33} & A_{34} & A_{35} \\ 0 & 0 & 0 & A_{44} & A_{45} \\ 0 & 0 & 0 & 0 & A_{55} \end{bmatrix}$$

This matrix represented as a FORTRAN array is

$$A = \begin{bmatrix} \times & \times & A_{13} & A_{24} & A_{35} \\ \times & A_{12} & A_{23} & A_{34} & A_{45} \\ A_{11} & A_{22} & A_{33} & A_{44} & A_{55} \end{bmatrix}$$

This corresponds to the CHARACTER*1 flags DIAGNL = 'N' and UPLO = 'U'. The matrix A^T is represented as the FORTRAN array

$$A = \begin{bmatrix} A_{11} & A_{22} & A_{33} & A_{44} & A_{55} \\ A_{12} & A_{23} & A_{34} & A_{45} & \times \\ A_{13} & A_{24} & A_{35} & \times & \times \end{bmatrix}$$

This corresponds to the `CHARACTER*1` flags `DIAGNL = 'N'` and `UPLO = 'L'`. In both examples, the entries indicated with an × are not referenced by IMSL subprograms.

Codiagonal Band Symmetric Storage Mode

This is an alternate storage mode for band symmetric matrices. It is not used by any of the BLAS, see Chapter 9, Programming Notes for BLAS. Storing data in a form transposed from the **Band Symmetric Storage Mode** maintains unit spacing between consecutive referenced array elements. This data structure is used to get good performance in the Cholesky decomposition algorithm that solves positive definite symmetric systems of linear equations $Ax = b$. The data type can be `REAL` or `DOUBLE PRECISION`. In the codiagonal band symmetric storage mode, the `NCODA` upper codiagonals and right-hand-side are stored in columns of this FORTRAN array. This array is declared by the following statement:

```
DIMENSION A(LDA, NCODA + 2)
```

The parameter `LDA` is the *leading positive dimension* of A. It must be at least as large as `N` + `NCODA`.

Consider a real symmetric 5 × 5 matrix with 2 codiagonals

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & 0 & 0 \\ A_{12} & A_{22} & A_{23} & A_{24} & 0 \\ A_{13} & A_{23} & A_{33} & A_{34} & A_{35} \\ 0 & A_{24} & A_{34} & A_{44} & A_{45} \\ 0 & 0 & A_{35} & A_{45} & A_{55} \end{bmatrix}$$

and a right-hand-side vector

$$b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix}$$

A FORTRAN declaration for the array to hold this matrix and right-hand-side vector is

```
PARAMETER (N = 5, NCODA = 2, LDA = N + NCODA)
REAL A(LDA, NCODA + 2)
```

The matrix and right-hand-side entries are placed in the FORTRAN array A as follows:

$$A = \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ A_{11} & \times & \times & b_1 \\ A_{22} & A_{12} & \times & b_2 \\ A_{33} & A_{23} & A_{13} & b_3 \\ A_{44} & A_{34} & A_{24} & b_4 \\ A_{55} & A_{45} & A_{35} & b_5 \end{bmatrix}$$

Entries marked with an × do not need to be defined. Certain of the IMSL band symmetric subprograms will initialize and use these values during the solution process. When a solution is computed, the b_i, $i = 1, \ldots, 5$, are replaced by x_i, $i = 1, \ldots, 5$.

The nonzero A_{ij}, $j \ge i$, are stored in array locations $A(j +$ NCODA, $(j - i) + 1)$. The right-hand-side entries b_j are stored in locations $A(j +$ NCODA, NCODA $+ 2)$. The solution entries x_j are returned in $A(j +$ NCODA, NCODA $+ 2)$.

Codiagonal Band Hermitian Storage Mode

This is an alternate storage mode for band Hermitian matrices. It is not used by any of the BLAS, see Chapter 9, Programming Notes for BLAS. In the codiagonal band Hermitian storage mode, the real and imaginary parts of the 2 * NCODA + 1 upper codiagonals and right-hand-side are stored in columns of a FORTRAN array. Note that there is no explicit use of the COMPLEX or the nonstandard data type DOUBLE COMPLEX data type in this storage mode.

For *Hermitian* complex matrices,

$$A = U + \sqrt{-1}V$$

where U and V are real matrices. They satisfy the conditions $U = U^T$ and $V = -V^T$. The right-hand-side

$$b = c + \sqrt{-1}\, d$$

where c and d are real vectors. The solution vector is denoted as

$$x = u + \sqrt{-1}v$$

where u and v are real. The storage is declared with the following statement

```
DIMENSION A(LDA, 2*NCODA + 3)
```

The parameter LDA is the *leading positive dimension* of A. It must be at least as large as N + NCODA.

The diagonal terms U_{jj} are stored in array locations A ($j +$ NCODA, 1). The diagonal V_{jj} are zero and are not stored. The nonzero U_{ij}, $j > i$, are stored in locations $A(j +$ NCODA, $2 * (j - i))$.

The nonzero V_{ij} are stored in locations $A(j +$ NCODA, $2*(j - i) + 1)$. The right side vector b is stored with c_j and d_j in locations $A(j +$ NCODA, 2*NCODA $+ 2)$ and $A(j +$ NCODA, 2*NCODA $+ 3)$ respectively. The real and imaginary parts of the solution, u_j and v_j, respectively overwrite c_j and d_j.

Consider a complex hermitian 5 × 5 matrix with 2 codiagonals

$$A=\begin{bmatrix} U_{11} & U_{12} & U_{13} & 0 & 0 \\ U_{12} & U_{22} & U_{23} & U_{24} & 0 \\ U_{13} & U_{23} & U_{33} & U_{34} & U_{35} \\ 0 & U_{24} & U_{34} & U_{44} & U_{45} \\ 0 & 0 & U_{35} & U_{45} & U_{55} \end{bmatrix} + \sqrt{-1}\begin{bmatrix} 0 & V_{12} & V_{13} & 0 & 0 \\ -V_{12} & 0 & V_{23} & V_{24} & 0 \\ -V_{13} & -V_{23} & 0 & V_{34} & V_{35} \\ 0 & -V_{24} & -V_{34} & 0 & V_{45} \\ 0 & 0 & -V_{35} & -V_{45} & 0 \end{bmatrix}$$

and a right-hand-side vector

$$b=\begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} + \sqrt{-1}\begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \end{bmatrix}$$

A FORTRAN declaration for the array to hold this matrix and right-hand-side vector is

```
PARAMETER (N = 5, NCODA = 2, LDA = N + NCODA)
REAL A(LDA,2*NCODA + 3)
```

The matrix and right-hand-side entries are placed in the FORTRAN array *A* as follows:

$$A=\begin{bmatrix} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times \\ U_{11} & \times & \times & \times & \times & c_1 & d_1 \\ U_{22} & U_{12} & V_{12} & \times & \times & c_2 & d_2 \\ U_{33} & U_{23} & V_{23} & U_{13} & V_{13} & c_3 & d_3 \\ U_{44} & U_{34} & V_{34} & U_{24} & V_{24} & c_4 & d_4 \\ U_{55} & U_{45} & V_{45} & U_{35} & V_{35} & c_5 & d_5 \end{bmatrix}$$

Entries marked with an × do not need to be defined.

Sparse Matrix Storage Mode

The sparse linear algebraic equation solvers in Chapter 1 accept the input matrix in *sparse storage mode*. This structure consists of `INTEGER` values `N` and `NZ`, the matrix dimension and the total number of nonzero entries in the matrix. In addition, there are two `INTEGER` arrays `IROW(*)` and `JCOL(*)` that contain unique matrix row and column coordinates where values are given. There is also an array *A*(`*`) of values. All other entries of the matrix are zero. Each of the arrays `IROW(*)`, `JCOL(*)`, *A*(`*`) must be of size `NZ`. The correspondence between matrix and array entries is given by

$$A_{\mathrm{IROW}(i),\mathrm{JCOL}(i)} = A(i),\, i = 1, \ldots, \mathrm{NZ}$$

The data type for *A*(`*`) can be one of `REAL`, `DOUBLE PRECISION`, or `COMPLEX`. If your FORTRAN compiler allows, the nonstandard data type `DOUBLE COMPLEX` can also be declared.

For example, consider a real 5 × 5 sparse matrix with 11 nonzero entries. The matrix A has the form

$$A = \begin{bmatrix} A_{11} & 0 & A_{13} & A_{14} & 0 \\ A_{21} & A_{22} & 0 & 0 & 0 \\ 0 & A_{32} & A_{33} & A_{34} & 0 \\ 0 & 0 & A_{43} & 0 & 0 \\ 0 & 0 & 0 & A_{54} & A_{55} \end{bmatrix}$$

Declarations of arrays and definitions of the values for this sparse matrix are

```
PARAMETER (NZ = 11, N = 5)
DIMENSION IROW(NZ), JCOL(NZ), A(NZ)
DATA IROW /1,1,1,2,2,3,3,3,4,5,5/
DATA JCOL /1,3,4,1,2,2,3,4,3,4,5/
```

DATA A $/A_{11}, A_{13}, A_{14}, A_{21}, A_{22}, A_{32}, A_{33}, A_{34},$ &
$A_{43}, A_{54}, A_{55}/$

Reserved Names

When writing programs accessing the MATH/LIBRARY, the user should choose FORTRAN names that do not conflict with names of IMSL subroutines, functions, or named common blocks, such as the workspace common block WORKSP (see page 1699). The user needs to be aware of two types of name conflicts that can arise. The first type of name conflict occurs when a name (technically a *symbolic name*) is not uniquely defined within a program unit (either a main program or a subprogram). For example, such a name conflict exists when the name RCURV is used to refer both to a type REAL variable and to the IMSL subroutine RCURV in a single program unit. Such errors are detected during compilation and are easy to correct. The second type of name conflict, which can be more serious, occurs when names of program units and named common blocks are not unique. For example, such a name conflict would be caused by the user defining a subroutine named WORKSP and also referencing an MATH/LIBRARY subroutine that uses the named common block WORKSP. Likewise, the user must not define a subprogram with the same name as a subprogram in the MATH/LIBRARY, that is referenced directly by the user's program or is referenced indirectly by other MATH/LIBRARY subprograms.

The MATH/LIBRARY consists of many routines, some that are described in the *User's Manual* and others that are not intended to be called by the user and, hence, that are not documented. If the choice of names were completely random over the set of valid FORTRAN names, and if a program uses only a small subset of the MATH/LIBRARY, the probability of name conflicts is very small. Since names are usually chosen to be mnemonic, however, the user may wish to take some precautions in choosing FORTRAN names.

Many IMSL names consist of a root name that may have a prefix to indicate the type of the routine. For example, the IMSL single precision subroutine for fitting a polynomial by least squares has the name RCURV, which is the root name, and the corresponding IMSL double precision routine has the name DRCURV. Associated with these two routines are R2URV and DR2URV. RCURV is listed in the Alphabetical Index of Routines, but DRCURV, R2URV, and DR2URV are not. The user of RCURV must consider both names RCURV and R2URV to be reserved; likewise, the user of DRCURV must consider both names DRCURV and DR2URV to be reserved. The root names of *all* routines and named common blocks that are used by the MATH/LIBRARY and that do not have a numeral in the second position of the root name are listed in the Alphabetical Index of Routines. Some of the routines in this Index (such as the "Level 2 BLAS") are not intended to be called by the user and so are not documented.

The careful user can avoid any conflicts with IMSL names if the following rules are observed:

- Do not choose a name that appears in the Alphabetical Summary of Routines in the *User's Manual*, nor one of these names preceded by a D, S_, D_, C_, or Z_.
- Do not choose a name of three or more characters with a numeral in the second or third position.

These simplified rules include many combinations that are, in fact, allowable. However, if the user selects names that conform to these rules, no conflict will be encountered.

Deprecated Features and Renamed Routines

Automatic Workspace Allocation

FORTRAN subroutines that work with arrays as input and output often require extra arrays for use as workspace while doing computations or moving around data. IMSL routines generally do not require the user explicitly to allocate such arrays for use as workspace. On most systems the workspace allocation is handled transparently. The only limitation is the actual amount of memory available on the system.

On some systems the workspace is allocated out of a stack that is passed as a FORTRAN array in a named common block WORKSP. A very similar use of a workspace stack is described by Fox et al. (1978, pages 116–121). (For compatiblity with older versions of the IMSL Libraries, space is allocated from the COMMON block, if possible.)

The arrays for workspace appear as arguments in lower-level routines. For example, the IMSL routine LSARG (in Chapter 1, "Linear Systems"), which solves systems of linear equations, needs arrays for workspace. LSARG allocates arrays from the common area, and passes them to the lower-level routine L2ARG which does the computations. In the "Comments" section of the documentation for LSARG, the amount of workspace is noted and the call to L2ARG is described. This scheme for using lower-level routines is followed throughout the IMSL Libraries. The names of these routines have a "2" in the second position (or in the third position in double precision routines having a "D" prefix). The user can provide workspace explicitly and call directly the "2-level" routine, which is documented along with the main routine. In a very few cases, the 2-level routine allows additional options that the main routine does not allow.

Prior to returning to the calling program, a routine that allocates workspace generally deallocates that space so that it becomes available for use in other routines.

Changing the Amount of Space Allocated

This section is relevant only to those systems on which the transparent workspace allocator is not available.

By default, the total amount of space allocated in the common area for storage of numeric data is 5000 numeric storage units. (A numeric storage unit is the amount of space required to store an integer or a real number. By comparison, a double precision unit is twice this amount. Therefore the total amount of space allocated in the common area for storage of numeric data is 2500 double precision units.) This space is allocated as needed for INTEGER, REAL, or other numeric data. For larger problems in which the default amount of workspace is insufficient, the user can change the allocation by supplying the FORTRAN statements to define the array in the named common block and by informing the IMSL workspace allocation system of the new size of the common array. To request 7000 units, the statements are

```
COMMON /WORKSP/ RWKSP
REAL RWKSP(7000)
CALL IWKIN(7000)
```

If an IMSL routine attempts to allocate workspace in excess of the amount available in the common stack, the routine issues a fatal error message that indicates how much space is needed and prints statements like those above to guide the user in allocating the necessary amount. The program below uses IMSL routine PERMA (see the Reference Material in this manual) to permute rows or columns of a matrix. This routine requires workspace equal to the number of columns, which in this example is too large. (Note that the work vector RWKSP must also provide extra space for bookkeeping.)

```
      USE_PERMA_INT
!                                 Specifications for local variables
      INTEGER    NRA, NCA, LDA, IPERMU(6000), IPATH
      REAL       A(2,6000)
!                                 Specifications for subroutines
!
      NRA = 2
      NCA = 6000
      LDA = 2
!                                 Initialize permutation index
      DO 10 I = 1, NCA
         IPERMU(I) = NCA + 1 - I
   10 CONTINUE
      IPATH = 2
      CALL PERMA (A, IPERMU, A, IPATH=IPATH)
      END
```

Output

```
*** TERMINAL ERROR 10 from PERMA.  Insufficient workspace for current
***          allocation(s). Correct by calling IWKIN from main program with
***          the three following statements:  (REGARDLESS OF PRECISION)
***                COMMON /WORKSP/  RWKSP
***                REAL RWKSP(6018)
***                CALL IWKIN(6018)

*** TERMINAL ERROR 10 from PERMA.  Workspace allocation was based on NCA =
***          6000.
```

In most cases, the amount of workspace is dependent on the parameters of the problem so the amount needed is known exactly. In a few cases, however, the amount of workspace is dependent on the data (for example, if it is necessary to count all of the unique values in a vector), so the

IMSL routine cannot tell in advance exactly how much workspace is needed. In such cases the error message printed is an estimate of the amount of space required.

Character Workspace

Since character arrays cannot be equivalenced with numeric arrays, a separate named common block WKSPCH is provided for character workspace. In most respects this stack is managed in the same way as the numeric stack. The default size of the character workspace is 2000 character units. (A character unit is the amount of space required to store one character.) The routine analogous to IWKIN used to change the default allocation is IWKCIN.

The routines in the following list are being deprecated in Version 2.0 of MATH/LIBRARY. A deprecated routine is one that is no longer used by anything in the library but is being included in the product for those users who may be currently referencing it in their application. However, any future versions of MATH/LIBRARY will not include these routines. If any of these routines are being called within an application, it is recommended that you change your code or retain the deprecated routine before replacing this library with the next version. Most of these routines were called by users only when they needed to set up their own workspace. Thus, the impact of these changes should be limited.

CZADD
CZINI
CZMUL
CZSTO
DE2AHF
DE2ASF
DE2BHF
DE2BSB
DE2BSF
DE2CCG
DE2CCH
DE2CHF
DE2CRG
DE2CRH
DE2CSB
DE2EHF
DE2ESB
DE2FHF
DE2FSB
DE2FSF
DE2LCG
DE2LCH
DE2LHF
DE2LRG
DE2LRH
DE2LSB
DE3CRG
DE3CRH
DE3LSF
DE4CRG
DE4ESF
DE5CRG
DE7CRG
DG2CCG
DG2CRG
DG2DF
DG2IND
DG2LCG
DG2LRG
DG3CCG
DG3DF
DG4CCG
DG5CCG
DG7CRG
DHOUAP
DHOUTR
DIVPBS
DN0ONF
DNCONF
DNCONG
E2ASF
E2AHF
E2BHF
E2BSB
E2BSF
E2CCG
E2CCH
E2CHF
E2CRG
E2CRH
E2CSB
E2EHF
E2ESB
E2FHF
E2FSB
E2FSF
E2LCG
E2LCH
E2LHF
E2LRG
E2LRH
E2LSB
E3CRG
E4CRG
E4ESF
E5CRG
E7CRG
G2CCG
G2CRG
G2LCG
G2LRG
G3CCG
G4CCG
G5CCG
G7CRG
N0ONF
NCONF
NCONG
SDADD
SDINI
SDMUL
SDSTO
SHOUAP
SHOUTR

The following routines have been renamed due to naming conflicts with other software manufacturers.

`CTIME` – replaced with `CPSEC`
`DTIME` – replaced with `TIMDY`
`PAGE` – replaced with `PGOPT`

Appendix A: GAMS Index

Description

This index lists routines in MATH/LIBRARY by a tree-structured classification scheme known as GAMS Version 2.0 (Boisvert, Howe, Kahaner, and Springmann (1990). Only the GAMS classes that contain MATH/LIBRARY routines are included in the index. The page number for the documentation and the purpose of the routine appear alongside the routine name.

The first level of the full classification scheme contains the following major subject areas:

A.	Arithmetic, Error Analysis
B.	Number Theory
C.	Elementary and Special Functions
D.	Linear Algebra
E.	Interpolation
F.	Solution of Nonlinear Equations
G.	Optimization
H.	Differentiation and Integration
I.	Differential and Integral Equations
J.	Integral Transforms
K.	Approximation
L.	Statistics, Probability
M.	Simulation, Stochastic Modeling
N.	Data Handling
O.	Symbolic Computation
P.	Computational Geometry
Q.	Graphics
R.	Service Routines
S.	Software Development Tools
Z.	Other

There are seven levels in the classification scheme. Classes in the first level are identified by a capital letter as is given above. Classes in the remaining levels are identified by alternating letter-and-number combinations. A single letter (a-z) is used with the odd-numbered levels. A number (1−26) is used within the even-numbered levels.

IMSL MATH/LIBRARY

A........... ARITHMETIC, ERROR ANALYSIS

A3......... Real

A3c....... Extended precision

`DQADD` Adds a double-precision scalar to the accumulator in extended precision.
`DQINI` Initializes an extended-precision accumulator with a double-precision scalar.
`DQMUL` Multiplies double-precision scalars in extended precision.
`DQSTO` Stores a double-precision approximation to an extended-precision scalar.

A4......... Complex

A4c....... Extended precision

`ZQADD` Adds a double complex scalar to the accumulator in extended precision.
`ZQINI` Initializes an extended-precision complex accumulator to a double complex scalar.
`ZQMUL` Multiplies double complex scalars using extended precision.
`ZQSTO` Stores a double complex approximation to an extended-precision complex scalar.

A6......... Change of representation

A6c....... Decomposition, construction

`PRIME` Decomposes an integer into its prime factors.

B........... NUMBER THEORY

`PRIME` Decomposes an integer into its prime factors.

C........... ELEMENTARY AND SPECIAL FUNCTIONS

C2......... Powers, roots, reciprocals

`HYPOT` Computes $\sqrt{a^2 + b^2}$ without underflow or overflow.

C19....... Other special functions

`CONST` Returns the value of various mathematical and physical constants.
`CUNIT` Converts `X` in units `XUNITS` to `Y` in units `YUNITS`.

D........... LINEAR ALGEBRA

D1......... Elementary vector and matrix operations

D1a....... Elementary vector operations

D1a1..... Set to constant

`CSET` Sets the components of a vector to a scalar, all complex.
`ISET` Sets the components of a vector to a scalar, all integer.

`SSET` Sets the components of a vector to a scalar, all single precision.

D1a2.....Minimum and maximum components

`ICAMAX` Finds the smallest index of the component of a complex vector having maximum magnitude.

`ICAMIN` Finds the smallest index of the component of a complex vector having minimum magnitude.

`IIMAX` Finds the smallest index of the maximum component of a integer vector.

`IIMIN` Finds the smallest index of the minimum of an integer vector.

`ISAMAX` Finds the smallest index of the component of a single-precision vector having maximum absolute value.

`ISAMIN` Finds the smallest index of the component of a single-precision vector having minimum absolute value.

`ISMAX` Finds the smallest index of the component of a single-precision vector having maximum value.

`ISMIN` Finds the smallest index of the component of a single-precision vector having minimum value.

D1a3.....Norm

D1a3a...L_1 (sum of magnitudes)

`DISL1` Computes the 1-norm distance between two points.

`SASUM` Sums the absolute values of the components of a single-precision vector.

`SCASUM` Sums the absolute values of the real part together with the absolute values of the imaginary part of the components of a complex vector.

D1a3b...L_2 (Euclidean norm)

`DISL2` Computes the Euclidean (2-norm) distance between two points.

`NORM2,CNORM2` Computes the Euclidean length of a vector or matrix, avoiding out-of-scale intermediate subexpressions.

`MNORM2,CMNORM2` Computes the Euclidean length of a vector or matrix, avoiding out-of-scale intermediate subexpressions

`NRM2, CNRM2` Computes the Euclidean length of a vector or matrix, avoiding out-of-scale intermediate subexpressions.

`SCNRM2` Computes the Euclidean norm of a complex vector.

`SNRM2` Computes the Euclidean length or L_2 norm of a single-precision vector.

D1a3c...L_∞ (maximum magnitude)

`DISLI` Computes the infinity norm distance between two points.

`ICAMAX` Finds the smallest index of the component of a complex vector having maximum magnitude.

`ISAMAX` Finds the smallest index of the component of a single-precision vector having maximum absolute value.

D1a4.....Dot product (inner product)

`CDOTC` Computes the complex conjugate dot product, $\bar{x}^T y$.

`CDOTU` Computes the complex dot product $x^T y$.

`CZCDOT` Computes the sum of a complex scalar plus a complex conjugate dot product, $a + \bar{x}^T y$, using a double-precision accumulator.

`CZDOTA` Computes the sum of a complex scalar, a complex dot product and the double-complex accumulator, which is set to the result $\texttt{ACC} \leftarrow \texttt{ACC} + a + x^T y$.

`CZDOTC` Computes the complex conjugate dot product, $\bar{x}^T y$, using a double-precision accumulator.

`CZDOTI` Computes the sum of a complex scalar plus a complex dot product using a double-complex accumulator, which is set to the result $\texttt{ACC} \leftarrow a + x^T y$.

`CZDOTU` Computes the complex dot product $x^T y$ using a double-precision accumulator.

`CZUDOT` Computes the sum of a complex scalar plus a complex dot product, $a + x^T y$, using a double-precision accumulator.

`DSDOT` Computes the single-precision dot product $x^T y$ using a double precision accumulator.

`SDDOTA` Computes the sum of a single-precision scalar, a single-precision dot product and the double-precision accumulator, which is set to the result $\texttt{ACC} \leftarrow \texttt{ACC} + a + x^T y$.

`SDDOTI` Computes the sum of a single-precision scalar plus a singleprecision dot product using a double-precision accumulator, which is set to the result $\texttt{ACC} \leftarrow a + x^T y$.

`SDOT` Computes the single-precision dot product $x^T y$.

`SDSDOT` Computes the sum of a single-precision scalar and a single precision dot product, $a + x^T y$, using a double-precision accumulator.

D1a5.....Copy or exchange (swap)

`CCOPY` Copies a vector x to a vector y, both complex.

`CSWAP` Interchanges vectors x and y, both complex.

`ICOPY` Copies a vector x to a vector y, both integer.

`ISWAP` Interchanges vectors x and y, both integer.

`SCOPY` Copies a vector x to a vector y, both single precision.

`SSWAP` Interchanges vectors x and y, both single precision.

D1a6.....Multiplication by scalar

`CSCAL` Multiplies a vector by a scalar, $y \leftarrow ay$, both complex.

`CSSCAL` Multiplies a complex vector by a single-precision scalar, $y \leftarrow ay$.

`CSVCAL` Multiplies a complex vector by a single-precision scalar and store the result in another complex vector, $y \leftarrow ax$.

`CVCAL` Multiplies a vector by a scalar and store the result in another vector, $y \leftarrow ax$, all complex.

`SSCAL` Multiplies a vector by a scalar, $y \leftarrow ay$, both single precision.

`SVCAL` Multiplies a vector by a scalar and store the result in another vector, $y \leftarrow ax$, all single precision.

D1a7..... Triad ($ax + y$ for vectors x, y and scalar a)

`CAXPY` Computes the scalar times a vector plus a vector, $y \leftarrow ax + y$, all complex.

`SAXPY` Computes the scalar times a vector plus a vector, $y \leftarrow ax + y$, all single precision.

D1a8..... Elementary rotation (Givens transformation) (*search also class D1b10*)

`CSROT` Applies a complex Givens plane rotation.

`CSROTM` Applies a complex modified Givens plane rotation.

`SROT` Applies a Givens plane rotation in single precision.

`SROTM` Applies a modified Givens plane rotation in single precision.

D1a10... Convolutions

`RCONV` Computes the convolution of two real vectors.

`VCONC` Computes the convolution of two complex vectors.

`VCONR` Computes the convolution of two real vectors.

D1a11... Other vector operations

`CADD` Adds a scalar to each component of a vector, $x \leftarrow x + a$, all complex.

`CSUB` Subtracts each component of a vector from a scalar, $x \leftarrow a - x$, all complex.

`DISL1` Computes the 1-norm distance between two points.

`DISL2` Computes the Euclidean (2-norm) distance between two points.

`DISLI` Computes the infinity norm distance between two points.

`IADD` Adds a scalar to each component of a vector, $x \leftarrow x + a$, all integer.

`ISUB` Subtracts each component of a vector from a scalar, $x \leftarrow a - x$, all integer.

`ISUM` Sums the values of an integer vector.

`SADD` Adds a scalar to each component of a vector, $x \leftarrow x + a$, all single precision.

`SHPROD` Computes the Hadamard product of two single-precision vectors.

`SPRDCT` Multiplies the components of a single-precision vector.

`SSUB` Subtracts each component of a vector from a scalar, $x \leftarrow a - x$, all single precision.

`SSUM` Sums the values of a single-precision vector.

`SXYZ` Computes a single-precision *xyz* product.

D1b.......Elementary matrix operations

`CGERC` Computes the rank-one update of a complex general matrix:
$A \leftarrow A + \alpha x\bar{y}^T$.

`CGERU` Computes the rank-one update of a complex general matrix:
$A \leftarrow A + \alpha xy^T$.

`CHER` Computes the rank-one update of an Hermitian matrix:
$A \leftarrow A + \alpha x\bar{x}^T$ with x complex and α real.

`CHER2` Computes a rank-two update of an Hermitian matrix:
$A \leftarrow A + \alpha x\bar{y}^T + \bar{\alpha} y\bar{x}^T$.

`CHER2K` Computes one of the Hermitian rank $2k$ operations:
$C \leftarrow \alpha A\bar{B}^T + \bar{\alpha} B\bar{A}^T + \beta C$ or $C \leftarrow \alpha\bar{A}^T B + \bar{\alpha}\bar{B}^T A + \beta C$,
where C is an n by n Hermitian matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

`CHERK` Computes one of the Hermitian rank k operations:
$C \leftarrow \alpha A\bar{A}^T + \beta C$ or $C \leftarrow \alpha\bar{A}^T A + \beta C$,
where C is an n by n Hermitian matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

`CSYR2K` Computes one of the symmetric rank $2k$ operations:
$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$ or $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$,
where C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

`CSYRK` Computes one of the symmetric rank k operations:
$C \leftarrow \alpha AA^T + \beta C$ or $C \leftarrow \alpha A^T A + \beta C$,
where C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

`CTBSV` Solves one of the complex triangular systems:
$x \leftarrow A^{-1}x, x \leftarrow \left(A^{-1}\right)^T x,$ or $x \leftarrow \left(\bar{A}^T\right)^{-1} x$,
where A is a triangular matrix in band storage mode.

`CTRSM` Solves one of the complex matrix equations:
$B \leftarrow \alpha A^{-1}B, B \leftarrow \alpha BA^{-1}, B \leftarrow \alpha\left(A^{-1}\right)^T B, B \leftarrow \alpha B\left(A^{-1}\right)^T$,

$B \leftarrow \alpha\left(\bar{A}^T\right)^{-1} B,$ or $B \leftarrow \alpha B\left(\bar{A}^T\right)^{-1}$
where A is a triangular matrix.

`CTRSV` Solves one of the complex triangular systems:
$x \leftarrow A^{-1}x, x \leftarrow \left(A^{-1}\right)^T x,$ or $x \leftarrow \left(\bar{A}^T\right)^{-1} x$,
where A is a triangular matrix.

HRRRR Computes the Hadamard product of two real rectangular matrices.

SGER Computes the rank-one update of a real general matrix: $A \leftarrow A + \alpha x y^T$.

SSYR Computes the rank-one update of a real symmetric matrix: $A \leftarrow A + \alpha x x^T$.

SSYR2 Computes the rank-two update of a real symmetric matrix: $A \leftarrow A + \alpha x y^T + \alpha y x^T$.

SSYR2K Computes one of the symmetric rank $2k$ operations: $C \leftarrow \alpha A B^T + \alpha B A^T + \beta C$ or $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$, where C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

SSYRK Computes one of the symmetric rank k operations: $C \leftarrow \alpha A A^T + \beta C$ or $C \leftarrow \alpha A^T A + \beta C$, where C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

STBSV Solves one of the triangular systems: $x \leftarrow A^{-1} x$ or $x \leftarrow \left(A^{-1}\right)^T x$, where A is a triangular matrix in band storage mode.

STRSM Solves one of the matrix equations: $B \leftarrow \alpha A^{-1} B$, $B \leftarrow \alpha B A^{-1}$, $B \leftarrow \alpha \left(A^{-1}\right)^T B$, or $B \leftarrow \alpha B \left(A^{-1}\right)^T$ where B is an m by n matrix and A is a triangular matrix.

STRSV Solves one of the triangular linear systems: $x \leftarrow A^{-1} x$ or $x \leftarrow \left(A^{-1}\right)^T x$, where A is a triangular matrix.

D1b2..... Norm

NR1CB Computes the 1-norm of a complex band matrix in band storage mode.

NR1RB Computes the 1-norm of a real band matrix in band storage mode.

NR1RR Computes the 1-norm of a real matrix.

NR2RR Computes the Frobenius norm of a real rectangular matrix.

NRIRR Computes the infinity norm of a real matrix.

D1b3..... Transpose

TRNRR Transposes a rectangular matrix.

D1b4 Multiplication by vector

BLINF Computes the bilinear form $x^T A y$.

CGBMV Computes one of the matrix-vector operations: $y \leftarrow \alpha A x + \beta y$, $y \leftarrow \alpha A^T x + \beta y$, or $y \leftarrow \alpha \overline{A}^T + \beta y$, where A is a matrix stored in band storage mode.

CGEMV Computes one of the matrix-vector operations:
$y \leftarrow \alpha Ax + \beta y,\ y \leftarrow \alpha A^T x + \beta y,$ or $y \leftarrow \alpha \overline{A}^T + \beta y$,

CHBMV Computes the matrix-vector operation
$y \leftarrow \alpha Ax + \beta y$,
where A is an Hermitian band matrix in band Hermitian storage.

CHEMV Computes the matrix-vector operation
$y \leftarrow \alpha Ax + \beta y$,
where A is an Hermitian matrix.

CTBMV Computes one of the matrix-vector operations:
$x \leftarrow Ax,\ x \leftarrow A^T x,$ or $x \leftarrow \overline{A}^T x$,
where A is a triangular matrix in band storage mode.

CTRMV Computes one of the matrix-vector operations:
$x \leftarrow Ax,\ x \leftarrow A^T x,$ or $x \leftarrow \overline{A}^T x$,
where A is a triangular matrix.

MUCBV Multiplies a complex band matrix in band storage mode by a complex vector.

MUCRV Multiplies a complex rectangular matrix by a complex vector.

MURBV Multiplies a real band matrix in band storage mode by a real vector.

MURRV Multiplies a real rectangular matrix by a vector.

SGBMV Computes one of the matrix-vector operations:
$y \leftarrow \alpha Ax + \beta y,$ or $y \leftarrow \alpha A^T x + \beta y$,
where A is a matrix stored in band storage mode.

SGEMV Computes one of the matrix-vector operations:
$y \leftarrow \alpha Ax + \beta y,$ or $y \leftarrow \alpha A^T x + \beta y$,

SSBMV Computes the matrix-vector operation
$y \leftarrow \alpha Ax + \beta y$,
where A is a symmetric matrix in band symmetric storage mode.

SSYMV Computes the matrix-vector operation
$y \leftarrow \alpha Ax + \beta y$,
where A is a symmetric matrix.

STBMV Computes one of the matrix-vector operations:
$x \leftarrow Ax$ or $x \leftarrow A^T x$,
where A is a triangular matrix in band storage mode.

STRMV Computes one of the matrix-vector operations:
$x \leftarrow Ax$ or $x \leftarrow A^T x$,
where A is a triangular matrix.

D1b5.....Addition, subtraction

ACBCB Adds two complex band matrices, both in band storage mode.

ARBRB Adds two band matrices, both in band storage mode.

D1b6.....Multiplication

CGEMM Computes one of the matrix-matrix operations:
$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha A^T B + \beta C, C \leftarrow \alpha AB^T$
$+\beta C, C \leftarrow \alpha A^T B^T + \beta C, C \leftarrow \alpha A\overline{B}^T + \beta C,$
or $C \leftarrow \alpha \overline{A}^T B + \beta C, C \leftarrow \alpha A^T \overline{B}^T + \beta C,$
$C \leftarrow \alpha \overline{A}^T B^T + \beta C,$ or $C \leftarrow \alpha \overline{A}^T \overline{B}^T + \beta C$

CHEMM Computes one of the matrix-matrix operations:
$C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$,
where *A* is an Hermitian matrix and *B* and *C* are *m* by *n* matrices.

CSYMM Computes one of the matrix-matrix operations:
$C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$,
where *A* is a symmetric matrix and *B* and *C* are *m* by *n* matrices.

CTRMM Computes one of the matrix-matrix operations:
$B \leftarrow \alpha AB, B \leftarrow \alpha A^T B, B \leftarrow \alpha BA, B \leftarrow \alpha BA^T,$
$B \leftarrow \alpha \overline{A}^T B,$ or $B \leftarrow \alpha B\overline{A}^T$
where *B* is an *m* by *n* matrix and *A* is a triangular matrix.

MCRCR Multiplies two complex rectangular matrices, *AB*.

MRRRR Multiplies two real rectangular matrices, *AB*.

MXTXF Computes the transpose product of a matrix, A^TA.

MXTYF Multiplies the transpose of matrix *A* by matrix *B*, A^TB.

MXYTF Multiplies a matrix *A* by the transpose of a matrix *B*, AB^T.

SGEMM Compute one of the matrix-matrix operations:
$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha A^T B + \beta C, C \leftarrow \alpha AB^T$.
$+\beta C,$ or $C \leftarrow \alpha A^T B^T + \beta C$

SSYMM Computes one of the matrix-matrix operations:
$C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$,
where *A* is a symmetric matrix and *B* and *C* are *m* by *n* matrices.

STRMM Computes one of the matrix-matrix operations:
$B \leftarrow \alpha AB, B \leftarrow \alpha A^T B$ or $B \leftarrow \alpha BA, B \leftarrow \alpha BA^T,$
where *B* is an *m* by *n* matrix and *A* is a triangular matrix.

D1b7.....Matrix polynomial

POLRG 1207 Evaluates a real general matrix polynomial.

D1b8.....Copy

CCBCB Copies a complex band matrix stored in complex band storage mode.

CCGCG Copies a complex general matrix.

CRBRB Copies a real band matrix stored in band storage mode.

CRGRG Copies a real general matrix.

D1b9..... Storage mode conversion

`CCBCG`	Converts a complex matrix in band storage mode to a complex matrix in full storage mode.
`CCGCB`	Converts a complex general matrix to a matrix in complex band storage mode.
`CHBCB`	Copies a complex Hermitian band matrix stored in band Hermitian storage mode to a complex band matrix stored in band storage mode.
`CHFCG`	Extends a complex Hermitian matrix defined in its upper triangle to its lower triangle.
`CRBCB`	Converts a real matrix in band storage mode to a complex matrix in band storage mode.
`CRBRG`	Converts a real matrix in band storage mode to a real general matrix.
`CRGCG`	Copies a real general matrix to a complex general matrix.
`CRGRB`	Converts a real general matrix to a matrix in band storage mode.
`CRRCR`	Copies a real rectangular matrix to a complex rectangular matrix.
`CSBRB`	Copies a real symmetric band matrix stored in band symmetric storage mode to a real band matrix stored in band storage mode.
`CSFRG`	Extends a real symmetric matrix defined in its upper triangle to its lower triangle.

D1b10... Elementary rotation (Givens transformation) (*search also class D1a8*)

`SROTG`	Constructs a Givens plane rotation in single precision.
`SROTMG`	Constructs a modified Givens plane rotation in single precision.

D2......... Solution of systems of linear equations (including inversion, *LU* and related decompositions)

D2a....... Real nonsymmetric matrices

`LSLTO`	Solves a real Toeplitz linear system.

D2a1..... General

`LFCRG`	Computes the *LU* factorization of a real general matrix and estimate its L_1 condition number.
`LFIRG`	Uses iterative refinement to improve the solution of a real general system of linear equations.
`LFSRG`	Solves a real general system of linear equations given the *LU* factorization of the coefficient matrix.
`LFTRG`	Computes the *LU* factorization of a real general matrix.
`LINRG`	Computes the inverse of a real general matrix.
`LSARG`	Solves a real general system of linear equations with iterative refinement.
`LSLRG`	Solves a real general system of linear equations without iterative refinement.
`LIN_SOL_GEN`	Solves a general system of linear equations $Ax = b$. Using optional arguments, any of several related computations

can be performed. These extra tasks include computing the *LU* factorization of *A* using partial pivoting, representing the determinant of *A*, computing the inverse matrix A^{-1}, and solving $A^T x = b$ or $Ax = b$ given the *LU* factorization of *A*.

D2a2..... Banded

LFCRB Computes the *LU* factorization of a real matrix in band storage mode and estimate its L_1 condition number.

LFIRB Uses iterative refinement to improve the solution of a real system of linear equations in band storage mode.

LFSRB Solves a real system of linear equations given the *LU* factorization of the coefficient matrix in band storage mode.

LFTRB Computes the *LU* factorization of a real matrix in band storage mode.

LSARB Solves a real system of linear equations in band storage mode with iterative refinement.

LSLRB Solves a real system of linear equations in band storage mode without iterative refinement.

STBSV Solves one of the triangular systems:

$x \leftarrow A^{-1}x$ or $x \leftarrow \left(A^{-1}\right)^T x$,

where *A* is a triangular matrix in band storage mode.

D2a2a ... Tridiagonal

LSLCR Computes the *LDU* factorization of a real tridiagonal matrix *A* using a cyclic reduction algorithm.

LSLTR Solves a real tridiagonal system of linear equations.

LIN_SOL_TRI Solves multiple systems of linear equations $A_j x_j = y_j, j = 1, \ldots, k$. Each matrix A_j is tridiagonal with the same dimension, *n*: The default solution method is based on *LU* factorization computed using cyclic reduction. An option is used to select Gaussian elimination with partial pivoting.

TRI_SOLVE A real, tri-diagonal, multiple system solver. Uses both cyclic reduction and Gauss elimination. Similar in function to `lin_sol_tri`.

D2a3..... Triangular

LFCRT Estimates the condition number of a real triangular matrix.

LINRT Computes the inverse of a real triangular matrix.

LSLRT Solves a real triangular system of linear equations.

STRSM Solves one of the matrix equations:

$B \leftarrow \alpha A^{-1}B$, $B \leftarrow \alpha BA^{-1}$, $B \leftarrow \alpha\left(A^{-1}\right)^T B$,

or $B \leftarrow \alpha B\left(A^{-1}\right)^T$

where *B* is an *m* by *n* matrix and *A* is a triangular matrix.

STRSV Solves one of the triangular linear systems:

$x \leftarrow A^{-1}x$ or $x \leftarrow \left(A^{-1}\right)^T x$

where A is a triangular matrix.

D2a4.....Sparse

LFSXG Solves a sparse system of linear equations given the *LU* factorization of the coefficient matrix.

LFTXG Computes the *LU* factorization of a real general sparse matrix.

LSLXG Solves a sparse system of linear algebraic equations by Gaussian elimination.

GMRES Uses restarted GMRES with reverse communication to generate an approximate solution of $Ax = b$.

D2b.......Real symmetric matrices

D2b1.....General

D2b1a...Indefinite

LCHRG Computes the Cholesky decomposition of a symmetric positive semidefinite matrix with optional column pivoting.

LFCSF Computes the UDU^T factorization of a real symmetric matrix and estimate its L_1 condition number.

LFISF Uses iterative refinement to improve the solution of a real symmetric system of linear equations.

LFSSF Solves a real symmetric system of linear equations given the UDU^T factorization of the coefficient matrix.

LFTSF Computes the UDU^T factorization of a real symmetric matrix.

LSASF Solves a real symmetric system of linear equations with iterative refinement.

LSLSF Solves a real symmetric system of linear equations without iterative refinement.

LIN_SOL_SELF Solves a system of linear equations $Ax = b$, where A is a self-adjoint matrix. Using optional arguments, any of several related computations can be performed. These extra tasks include computing and saving the factorization of A using symmetric pivoting, representing the determinant of A, computing the inverse matrix A^{-1}, or computing the solution of $Ax = b$ given the factorization of A. An optional argument is provided indicating that A is positive definite so that the Cholesky decomposition can be used.

D2b1b...Positive definite

LCHRG Computes the Cholesky decomposition of a symmetric positive semidefinite matrix with optional column pivoting.

LFCDS Computes the $R^T R$ Cholesky factorization of a real symmetric positive definite matrix and estimate its L_1 condition number.

LFIDS Uses iterative refinement to improve the solution of a real symmetric positive definite system of linear equations.

LFSDS Solves a real symmetric positive definite system of linear equations given the $R^T R$ Choleksy factorization of the coefficient matrix.

LFTDS Computes the $R^T R$ Cholesky factorization of a real symmetric positive definite matrix.

LINDS Computes the inverse of a real symmetric positive definite matrix.

LSADS Solves a real symmetric positive definite system of linear equations with iterative refinement.

LSLDS Solves a real symmetric positive definite system of linear equations without iterative refinement.

LIN_SOL_SELF Solves a system of linear equations $Ax = b$, where A is a self-adjoint matrix. Using optional arguments, any of several related computations can be performed. These extra tasks include computing and saving the factorization of A using symmetric pivoting, representing the determinant of A, computing the inverse matrix A^{-1}, or computing the solution of $Ax = b$ given the factorization of A. An optional argument is provided indicating that A is positive definite so that the Cholesky decomposition can be used.

D2b2.....Positive definite banded

LFCQS Computes the $R^T R$ Cholesky factorization of a real symmetric positive definite matrix in band symmetric storage mode and estimate its L_1 condition number.

LFDQS Computes the determinant of a real symmetric positive definite matrix given the $R^T R$ Cholesky factorization of the band symmetric storage mode.

LFIQS Uses iterative refinement to improve the solution of a real symmetric positive definite system of linear equations in band symmetric storage mode.

LFSQS Solves a real symmetric positive definite system of linear equations given the factorization of the coefficient matrix in band symmetric storage mode.

LFTQS Computes the $R^T R$ Cholesky factorization of a real symmetric positive definite matrix in band symmetric storage mode.

LSAQS Solves a real symmetric positive definite system of linear equations in band symmetric storage mode with iterative refinement.

LSLPB Computes the $R^T DR$ Cholesky factorization of a real symmetric positive definite matrix A in codiagonal band symmetric storage mode. Solve a system $Ax = b$.

LSLQS Solves a real symmetric positive definite system of linear equations in band symmetric storage mode without iterative refinement.

D2b4..... Sparse

JCGRC Solves a real symmetric definite linear system using the Jacobi preconditioned conjugate gradient method with reverse communication.

LFSXD Solves a real sparse symmetric positive definite system of linear equations, given the Cholesky factorization of the coefficient matrix.

LNFXD Computes the numerical Cholesky factorization of a sparse symmetrical matrix A.

LSCXD Performs the symbolic Cholesky factorization for a sparse symmetric matrix using a minimum degree ordering or a userspecified ordering, and set up the data structure for the numerical Cholesky factorization.

LSLXD Solves a sparse system of symmetric positive definite linear algebraic equations by Gaussian elimination.

PCGRC Solves a real symmetric definite linear system using a preconditioned conjugate gradient method with reverse communication.

D2c....... Complex non-Hermitian matrices

LSLCC Solves a complex circulant linear system.

LSLTC Solves a complex Toeplitz linear system.

D2c1..... General

LFCCG Computes the LU factorization of a complex general matrix and estimate its L_1 condition number.

LFICG Uses iterative refinement to improve the solution of a complex general system of linear equations.

LFSCG Solves a complex general system of linear equations given the LU factorization of the coefficient matrix.

LFTCG Computes the LU factorization of a complex general matrix.

LINCG Computes the inverse of a complex general matrix.

LSACG Solves a complex general system of linear equations with iterative refinement.

LSLCG Solves a complex general system of linear equations without iterative refinement.

LIN_SOL_GEN Solves a general system of linear equations $Ax = b$. Using optional arguments, any of several related computations can be performed. These extra tasks include computing the LU factorization of A using partial pivoting, representing the determinant of A, computing the inverse matrix A^{-1}, and solving $A^Tx = b$ or $Ax = b$ given the LU factorization of A.

D2c2..... Banded

`CTBSV` Solves one of the complex triangular systems:

$$x \leftarrow A^{-1}x\,,\ x \leftarrow \left(A^{-1}\right)^T x,\ \text{or}\ x \leftarrow \left(\overline{A}^T\right)^{-1} x\,,$$

where A is a triangular matrix in band storage mode.

`LFCCB` Computes the LU factorization of a complex matrix in band storage mode and estimate its L_1 condition number.

`LFICB` Uses iterative refinement to improve the solution of a complex system of linear equations in band storage mode.

`LFSCB` Solves a complex system of linear equations given the LU factorization of the coefficient matrix in band storage mode.

`LFTCB` Computes the LU factorization of a complex matrix in band storage mode.

`LSACB` Solves a complex system of linear equations in band storage mode with iterative refinement.

`LSLCB` Solves a complex system of linear equations in band storage mode without iterative refinement.

D2c2a ... Tridiagonal

`LSLCQ` Computes the LDU factorization of a complex tridiagonal matrix A using a cyclic reduction algorithm.

`LSLTQ` Solves a complex tridiagonal system of linear equations.

`LIN_SOL_TRI` Solves multiple systems of linear equations $A_jx_j = y_j$, $j = 1$, …, k. Each matrix A_j is tridiagonal with the same dimension, n: The default solution method is based on LU factorization computed using cyclic reduction. An option is used to select Gaussian elimination with partial pivoting.

D2c3..... Triangular

`CTRSM` Solves one of the complex matrix equations:

$$B \leftarrow \alpha A^{-1}B,\ B \leftarrow \alpha BA^{-1},\ B \leftarrow \alpha\left(A^{-1}\right)^T B,\ B \leftarrow \alpha B\left(A^{-1}\right)^T,$$

$$B \leftarrow \alpha\left(\overline{A}^T\right)^{-1} B, \text{or}\ B \leftarrow \alpha B\left(\overline{A}^T\right)^{-1}$$

where A is a traiangular matrix.

`CTRSV` Solves one of the complex triangular systems:

$$x \leftarrow A^{-1}x\,,\ x \leftarrow \left(A^{-1}\right)^T x,\ \text{or}\ x \leftarrow \left(\overline{A}^T\right)^{-1} x$$

where A is a triangular matrix.

`LFCCT` Estimates the condition number of a complex triangular matrix.

`LINCT` Computes the inverse of a complex triangular matrix.

`LSLCT` Solves a complex triangular system of linear equations.

D2c4.....Sparse

`LFSZG` Solves a complex sparse system of linear equations given the *LU* factorization of the coefficient matrix.

`LFTZG` Computes the *LU* factorization of a complex general sparse matrix.

`LSLZG` Solves a complex sparse system of linear equations by Gaussian elimination.

D2d.......Complex Hermitian matrices

D2d1.....General

D2d1a...Indefinite

`LFCHF` Computes the $U DU^H$ factorization of a complex Hermitian matrix and estimate its L_1 condition number.

`LFDHF` Computes the determinant of a complex Hermitian matrix given the $U DU^H$ factorization of the matrix.

`LFIHF` Uses iterative refinement to improve the solution of a complex Hermitian system of linear equations.

`LFSHF` Solves a complex Hermitian system of linear equations given the $U DU^H$ factorization of the coefficient matrix.

`LFTHF` Computes the $U DU^H$ factorization of a complex Hermitian matrix.

`LSAHF` Solves a complex Hermitian system of linear equations with iterative refinement.

`LSLHF` Solves a complex Hermitian system of linear equations without iterative refinement.

`LIN_SOL_SELF` Solves a system of linear equations $Ax = b$, where A is a self-adjoint matrix. Using optional arguments, any of several related computations can be performed. These extra tasks include computing and saving the factorization of A using symmetric pivoting, representing the determinant of A, computing the inverse matrix A^{-1}, or computing the solution of $Ax = b$ given the factorization of A. An optional argument is provided indicating that A is positive definite so that the Cholesky decomposition can be used.

D2d1b...Positive definite

`LFCDH` Computes the $R^H R$ factorization of a complex Hermitian positive definite matrix and estimate its L_1 condition number.

`LFIDH` Uses iterative refinement to improve the solution of a complex Hermitian positive definite system of linear equations.

`LFSDH` Solves a complex Hermitian positive definite system of linear equations given the $R^H R$ factorization of the coefficient matrix.

`LFTDH` Computes the $R^H R$ factorization of a complex Hermitian positive definite matrix.

`LSADH` Solves a Hermitian positive definite system of linear equations with iterative refinement.

`LSLDH` Solves a complex Hermitian positive definite system of linear equations without iterative refinement.

`LIN_SOL_SELF` Solves a system of linear equations $Ax = b$, where A is a self-adjoint matrix. Using optional arguments, any of several related computations can be performed. These extra tasks include computing and saving the factorization of A using symmetric pivoting, representing the determinant of A, computing the inverse matrix A^{-1}, or computing the solution of $Ax = b$ given the factorization of A. An optional argument is provided indicating that A is positive definite so that the Cholesky decomposition can be used.

D2d2.....Positive definite banded

`LFCQH` Computes the $R^H R$ factorization of a complex Hermitian positive definite matrix in band Hermitian storage mode and estimate its L_1 condition number.

`LFIQH` Uses iterative refinement to improve the solution of a complex Hermitian positive definite system of linear equations in band Hermitian storage mode.

`LFSQH` Solves a complex Hermitian positive definite system of linear equations given the factorization of the coefficient matrix in band Hermitian storage mode.

`LFTQH` Computes the $R^H R$ factorization of a complex Hermitian positive definite matrix in band Hermitian storage mode.

`LSAQH` Solves a complex Hermitian positive definite system of linear equations in band Hermitian storage mode with iterative refinement.

`LSLQB` Computes the $R^H DR$ Cholesky factorization of a complex hermitian positive-definite matrix A in codiagonal band hermitian storage mode. Solve a system $Ax = b$.

`LSLQH` Solves a complex Hermitian positive definite system of linearequations in band Hermitian storage mode without iterative refinement.

D2d4.....Sparse

`LFSZD` Solves a complex sparse Hermitian positive definite system of linear equations, given the Cholesky factorization of the coefficient matrix.

`LNFZD` Computes the numerical Cholesky factorization of a sparse Hermitian matrix A.

`LSLZD` Solves a complex sparse Hermitian positive definite system of linear equations by Gaussian elimination.

D3.........Determinants

D3a.......Real nonsymmetric matrices

D3a1.....General

`LFDRG`	Computes the determinant of a real general matrix given the *LU* factorization of the matrix.

D3a2.....Banded

`LFDRB`	Computes the determinant of a real matrix in band storage mode given the *LU* factorization of the matrix.

D3a3.....Triangular

`LFDRT`	Computes the determinant of a real triangular matrix.

D3b.......Real symmetric matrices

D3b1.....General

D3b1a...Indefinite

`LFDSF`	Computes the determinant of a real symmetric matrix given the $U\,DU^T$ factorization of the matrix.

D3b1b...Positive definite

`LFDDS`	Computes the determinant of a real symmetric positive definite matrix given the $R^H\,R$ Cholesky factorization of the matrix.

D3c.......Complex non-Hermitian matrices

D3c1.....General

`LFDCG`	Computes the determinant of a complex general matrix given the *LU* factorization of the matrix.

D3c2.....Banded

`LFDCB`	Computes the determinant of a complex matrix given the *LU* factorization of the matrix in band storage mode.

D3c3.....Triangular

`LFDCT`	Computes the determinant of a complex triangular matrix.

D3d.......Complex Hermitian matrices

D3d1.....General

D3d1b...Positive definite

`LFDDH`	Computes the determinant of a complex Hermitian positive definite matrix given the $R^H\,R$ Cholesky factorization of the matrix.

D3d2.....Positive definite banded

`LFDQH`	Computes the determinant of a complex Hermitian positive definite matrix given the $R^H\,R$ Cholesky factorization in band Hermitian storage mode.

D4.........Eigenvalues, eigenvectors

D4a.......Ordinary eigenvalue problems ($Ax = \lambda x$)

D4a1.....Real symmetric

`EVASF` Computes the largest or smallest eigenvalues of a real symmetric matrix.

`EVBSF` Computes selected eigenvalues of a real symmetric matrix.

`EVCSF` Computes all of the eigenvalues and eigenvectors of a real symmetric matrix.

`EVESF` Computes the largest or smallest eigenvalues and the corresponding eigenvectors of a real symmetric matrix.

`EVFSF` Computes selected eigenvalues and eigenvectors of a real symmetric matrix.

`EVLSF` Computes all of the eigenvalues of a real symmetric matrix.

`LIN_EIG_SELF` Computes the eigenvalues of a self-adjoint matrix, A. Optionally, the eigenvectors can be computed. This gives the decomposition $A = VDV^T$, where V is an $n \times n$ orthogonal matrix and D is a real diagonal matrix.

D4a2.....Real nonsymmetric

`EVCRG` Computes all of the eigenvalues and eigenvectors of a real matrix.

`EVLRG` Computes all of the eigenvalues of a real matrix.

`LIN_EIG_GEN` Computes the eigenvalues of an $n \times n$ matrix, A. Optionally, the eigenvectors of A or A^T are computed. Using the eigenvectors of A gives the decomposition $AV = VE$, where V is an $n \times n$ complex matrix of eigenvectors, and E is the complex diagonal matrix of eigenvalues. Other options include the reduction of A to upper triangular or Schur form, reduction to block upper triangular form with 2×2 or unit sized diagonal block matrices, and reduction to upper Hessenberg form.

D4a3.....Complex Hermitian

`EVAHF` Computes the largest or smallest eigenvalues of a complex Hermitian matrix.

`EVBHF` Computes the eigenvalues in a given range of a complex Hermitian matrix.

`EVCHF` Computes all of the eigenvalues and eigenvectors of a complex Hermitian matrix.

`EVEHF` Computes the largest or smallest eigenvalues and the corresponding eigenvectors of a complex Hermitian matrix.

`EVFHF` Computes the eigenvalues in a given range and the corresponding eigenvectors of a complex Hermitian matrix.

`EVLHF` Computes all of the eigenvalues of a complex Hermitian matrix.

`LIN_EIG_SELF` Computes the eigenvalues of a self-adjoint matrix, A. Optionally, the eigenvectors can be computed. This gives the decomposition $A = VDV^T$, where V is an $n \times n$ orthogonal matrix and D is a real diagonal matrix.

D4a4.....Complex non-Hermitian

`EVCCG` Computes all of the eigenvalues and eigenvectors of a complex matrix.

`EVLCG` Computes all of the eigenvalues of a complex matrix.

`LIN_EIG_GEN` Computes the eigenvalues of an $n \times n$ matrix, A. Optionally, the eigenvectors of A or A^T are computed. Using the eigenvectors of A gives the decomposition $AV = VE$, where V is an $n \times n$ complex matrix of eigenvectors, and E is the complex diagonal matrix of eigenvalues. Other options include the reduction of A to upper triangular or Schur form, reduction to block upper triangular form with 2 × 2 or unit sized diagonal block matrices, and reduction to upper Hessenberg form.

D4a6.....Banded

`EVASB` Computes the largest or smallest eigenvalues of a real symmetric matrix in band symmetric storage mode.

`EVBSB` Computes the eigenvalues in a given interval of a real symmetric matrix stored in band symmetric storage mode.

`EVCSB` Computes all of the eigenvalues and eigenvectors of a real symmetric matrix in band symmetric storage mode.

`EVESB` Computes the largest or smallest eigenvalues and the corresponding eigenvectors of a real symmetric matrix in band symmetric storage mode.

`EVFSB` Computes the eigenvalues in a given interval and the corresponding eigenvectors of a real symmetric matrix stored in band symmetric storage mode.

`EVLSB` Computes all of the eigenvalues of a real symmetric matrix in band symmetric storage mode.

D4b.......Generalized eigenvalue problems (e.g., $Ax = \lambda Bx$)

D4b1.....Real symmetric

`GVCSP` Computes all of the eigenvalues and eigenvectors of the generalized real symmetric eigenvalue problem $Az = \lambda Bz$, with B symmetric positive definite.

`GVLSP` Computes all of the eigenvalues of the generalized real symmetric eigenvalue problem $Az = \lambda Bz$, with B symmetric positive definite.

`LIN_GEIG_GEN` Computes the generalized eigenvalues of an $n \times n$ matrix pencil, $Av \cong \lambda Bv$. Optionally, the generalized eigenvectors are computed. If either of A or B is nonsingular, there are diagonal matrices α and β and a complex matrix V computed such that $AV\beta = BV\alpha$.

D4b2.....Real general

GVCRG Computes all of the eigenvalues and eigenvectors of a generalized real eigensystem $Az = \lambda Bz$.

GVLRG Computes all of the eigenvalues of a generalized real eigensystem $Az = \lambda Bz$.

LIN_GEIG_GEN Computes the generalized eigenvalues of an $n \times n$ matrix pencil, $Av \cong \lambda Bv$. Optionally, the generalized eigenvectors are computed. If either of A or B is nonsingular, there are diagonal matrices α and β and a complex matrix V computed such that $AV\beta = BV\alpha$.

D4b4.....Complex general

GVCCG Computes all of the eigenvalues and eigenvectors of a generalized complex eigensystem $Az = \lambda Bz$.

GVLCG Computes all of the eigenvalues of a generalized complex eigensystem $Az = \lambda Bz$.

LIN_GEIG_GEN Computes the generalized eigenvalues of an $n \times n$ matrix pencil, $Av \cong \lambda Bv$. Optionally, the generalized eigenvectors are computed. If either of A or B is nonsingular, there are diagonal matrices α and β and a complex matrix V computed such that $AV\beta = BV\alpha$.

D4c.......Associated operations

BALANC, CBSLANC Balances a general matrix before computing the eigenvalue-eigenvector decomposition.

EPICG Computes the performance index for a complex eigensystem.

EPIHF Computes the performance index for a complex Hermitian eigensystem.

EPIRG Computes the performance index for a real eigensystem.

EPISB Computes the performance index for a real symmetric eigensystem in band symmetric storage mode.

EPISF Computes the performance index for a real symmetric eigensystem.

GPICG Computes the performance index for a generalized complex eigensystem $Az = \lambda Bz$.

GPIRG Computes the performance index for a generalized real eigensystem $Az = \lambda Bz$.

GPISP Computes the performance index for a generalized real symmetric eigensystem problem.

PERFECT_SHIFT Computes eigenvectors using actual eigenvalue as an explicit shift. Called by lin_eig_self.

PWK A rational QR algorithm for computing eigenvalues of real, symmetric tri-diagonal matrices. Called by lin_svd and lin_eig_self.

D4c2.....Compute eigenvalues of matrix in compact form

D4c2b...Hessenberg

EVCCH Computes all of the eigenvalues and eigenvectors of a complex upper Hessenberg matrix.

`EVCRH` Computes all of the eigenvalues and eigenvectors of a real upper Hessenberg matrix.

`EVLCH` Computes all of the eigenvalues of a complex upper Hessenberg matrix.

`EVLRH` Computes all of the eigenvalues of a real upper Hessenberg matrix.

D5.........*QR* decomposition, Gram-Schmidt orthogonalization

`LQERR` Accumulates the orthogonal matrix *Q* from its factored form given the *QR* factorization of a rectangular matrix *A*.

`LQRRR` Computes the *QR* decomposition, $AP = QR$, using Householder transformations.

`LQRSL` Computes the coordinate transformation, projection, and complete the solution of the least-squares problem $Ax = b$.

`LSBRR` Solves a linear least-squares problem with iterative refinement.

`LSQRR` Solves a linear least-squares problem without iterative refinement.

D6.........Singular value decomposition

`LSVCR` Computes the singular value decomposition of a complex matrix.

`LSVRR` Computes the singular value decomposition of a real matrix.

`LIN_SOL_SVD` Solves a rectangular least-squares system of linear equations $Ax \cong b$ using singular value decomposition, $A = USV^T$. Using optional arguments, any of several related computations can be performed. These extra tasks include computing the rank of *A*, the orthogonal $m \times m$ and $n \times n$ matrices *U* and *V*, and the $m \times n$ diagonal matrix of singular values, *S*.

`LIN_SVD` Computes the singular value decomposition (SVD) of a rectangular matrix, *A*. This gives the decomposition $A = USV^T$, where *V* is an $n \times n$ orthogonal matrix, *U* is an $m \times m$ orthogonal matrix, and *S* is a real, rectangular diagonal matrix.

D7.........Update matrix decompositions

D7b.......Cholesky

`LDNCH` Downdates the R^TR Cholesky factorization of a real symmetric positive definite matrix after a rank-one matrix is removed.

`LUPCH` Updates the R^TR Cholesky factorization of a real symmetric positive definite matrix after a rank-one matrix is added.

D7c.......*QR*

LUPQR Computes an updated QR factorization after the rank-one matrix αxy^T is added.

D9......... Singular, overdetermined or underdetermined systems of linear equations, generalized inverses

D9a....... Unconstrained

D9a1..... Least squares (L_2) solution

BAND_
ACCUMALATION Accumulatez and solves banded least-squares problem using Householder transformations.

BAND_SOLVE Accumulatez and solves banded least-squares problem using Householder transformations.

HOUSE_HOLDER Accumulates and solves banded least-squares problem using Householder transformations.

LQRRR Computes the QR decomposition, $AP = QR$, using Householder transformations.

LQRRV Computes the least-squares solution using Householder transformations applied in blocked form.

LQRSL Computes the coordinate transformation, projection, and complete the solution of the least-squares problem $Ax = b$.

LSBRR Solves a linear least-squares problem with iterative refinement.

LSQRR Solves a linear least-squares problem without iterative refinement.

LIN_SOL_LSQ Solves a rectangular system of linear equations $Ax \cong b$, in a least-squares sense. Using optional arguments, any of several related computations can be performed. These extra tasks include computing and saving the factorization of A using column and row pivoting, representing the determinant of A, computing the generalized inverse matrix $A^\dagger$, or computing the least-squares solution of $Ax \cong b$ or $A^Ty \cong d$ given the factorization of A. An optional argument is provided for computing the following unscaled covariance matrix: $C = (A^TA)^{-1}$.

LIN_SOL_SVD Solves a rectangular least-squares system of linear equations $Ax \cong b$ using singular value decomposition, $A = USV^T$. Using optional arguments, any of several related computations can be performed. These extra tasks include computing the rank of A, the orthogonal $m \times m$ and $n \times n$ matrices U and V, and the $m \times n$ diagonal matrix of singular values, S.

D9b....... Constrained

D9b1..... Least squares (L_2) solution

LCLSQ Solves a linear least-squares problem with linear constraints.

D9c. ...... Generalized inverses

LSGRR Computes the generalized inverse of a real matrix.

LIN_SOL_LSQ Solves a rectangular system of linear equations $Ax \cong b$, in a least-squares sense. Using optional arguments, any of several related computations can be performed. These extra tasks include computing and saving the factorization of A using column and row pivoting, representing the determinant of A, computing the generalized inverse matrix $A^{\dagger}$, or computing the least-squares solution of $Ax \cong b$ or $A^T y \cong d$ given the factorization of A. An optional argument is provided for computing the following unscaled covariance matrix: $C = (A^T A)^{-1}$.

E .... INTERPOLATION

E1 .... Univariate data (curve fitting)

E1a .... Polynomial splines (piecewise polynomials)

BSINT Computes the spline interpolant, returning the B-spline coefficients.

CSAKM Computes the Akima cubic spline interpolant.

CSCON Computes a cubic spline interpolant that is consistent with the concavity of the data.

CSDEC Computes the cubic spline interpolant with specified derivative endpoint conditions.

CSHER Computes the Hermite cubic spline interpolant.

CSIEZ Computes the cubic spline interpolant with the ‘not-a-knot’ condition and return values of the interpolant at specified points.

CSINT Computes the cubic spline interpolant with the ‘not-a-knot’ condition.

CSPER Computes the cubic spline interpolant with periodic boundary conditions.

QDVAL Evaluates a function defined on a set of points using quadratic interpolation.

SPLEZ Computes the values of a spline that either interpolates or fits user-supplied data.

SPLINE_FITTING Solves constrained least-squares fitting of one-dimensional data by B-splines.

SPlINE_SUPPORT B-spline function and derivative evaluation package.

E2 .... Multivariate data (surface fitting)

E2a .... Gridded

BS2IN Computes a two-dimensional tensor-product spline interpolant, returning the tensor-product B-spline coefficients.

BS3IN Computes a three-dimensional tensor-product spline interpolant, returning the tensor-product B-spline coefficients.

QD2DR Evaluates the derivative of a function defined on a rectangular grid using quadratic interpolation.

QD2VL Evaluates a function defined on a rectangular grid using quadratic interpolation.

QD3DR Evaluates the derivative of a function defined on a rectangular three-dimensional grid using quadratic interpolation.

QD3VL Evaluates a function defined on a rectangular three-dimensional grid using quadratic interpolation.

SURFACE_FITTING Solves constrained least-squares fitting of two-dimensional data by tensor products of B-splines.

E2b....... Scattered

SURF Computes a smooth bivariate interpolant to scattered data that is locally a quintic polynomial in two variables.

SURFACE_FAIRING Constrained weighted least-squares fitting of tensor product B-splines to discrete data, with covariance matrix and constraints at points.

E3......... Service routines for interpolation

E3a....... Evaluation of fitted functions, including quadrature

E3a1..... Function evaluation

BS1GD Evaluates the derivative of a spline on a grid, given its B-spline representation.

BS2DR Evaluates the derivative of a two-dimensional tensor-product spline, given its tensor-product B-spline representation.

BS2GD Evaluates the derivative of a two-dimensional tensor-product spline, given its tensor-product B-spline representation on a grid.

BS2VL Evaluates a two-dimensional tensor-product spline, given its tensor-product B-spline representation.

BS3GD Evaluates the derivative of a three-dimensional tensor-product spline, given its tensor-product B-spline representation on a grid.

BS3VL Evaluates a three-dimensional tensor-product spline, given its tensor-product B-spline representation.

BSVAL Evaluates a spline, given its B-spline representation.

CSVAL Evaluates a cubic spline.

PPVAL Evaluates a piecewise polynomial.

QDDER Evaluates the derivative of a function defined on a set of points using quadratic interpolation.

E3a2..... Derivative evaluation

BS1GD Evaluates the derivative of a spline on a grid, given its B-spline representation.

BS2DR Evaluates the derivative of a two-dimensional tensor-product spline, given its tensor-product B-spline representation.

`BS2GD` Evaluates the derivative of a two-dimensional tensor-product spline, given its tensor-product B-spline representation on a grid.

`BS3DR` Evaluates the derivative of a three-dimensional tensor-product spline, given its tensor-product B-spline representation.

`BS3GD` Evaluates the derivative of a three-dimensional tensor-product spline, given its tensor-product B-spline representation on a grid.

`BSDER` Evaluates the derivative of a spline, given its B-spline representation.

`CS1GD` Evaluates the derivative of a cubic spline on a grid.

`CSDER` Evaluates the derivative of a cubic spline.

`PP1GD` Evaluates the derivative of a piecewise polynomial on a grid.

`PPDER` Evaluates the derivative of a piecewise polynomial.

`QDDER` Evaluates the derivative of a function defined on a set of points using quadratic interpolation.

E3a3 ..... Quadrature

`BS2IG` Evaluates the integral of a tensor-product spline on a rectangular domain, given its tensor-product B-spline representation.

`BS3IG` Evaluates the integral of a tensor-product spline in three dimensions over a three-dimensional rectangle, given its tensorproduct B-spline representation.

`BSITG` Evaluates the integral of a spline, given its B-spline representation.

`CSITG` Evaluates the integral of a cubic spline.

E3b .... Grid or knot generation

`BSNAK` Computes the ‘not-a-knot’ spline knot sequence.

`BSOPK` Computes the ‘optimal’ spline knot sequence.

E3c .... Manipulation of basis functions (e.g., evaluation, change of basis)

`BSCPP` Converts a spline in B-spline representation to piecewise polynomial representation.

F .... SOLUTION OF NONLINEAR EQUATIONS

F1 .... Single equation

F1a........ Polynomial

F1a1...... Real coefficients

`ZPLRC` Finds the zeros of a polynomial with real coefficients using Laguerre’s method.

`ZPORC` Finds the zeros of a polynomial with real coefficients using the Jenkins-Traub three-stage algorithm.

F1a2...... Complex coefficients

`ZPOCC` Finds the zeros of a polynomial with complex coefficients using the Jenkins-Traub three-stage algorithm.

F1b .... Nonpolynomial

`ZANLY` Finds the zeros of a univariate complex function using Müller's method.

`ZBREN` Finds a zero of a real function that changes sign in a given interval.

`ZREAL` Finds the real zeros of a real function using Müller's method.

F2 .... System of equations

`NEQBF` Solves a system of nonlinear equations using factored secant update with a finite-difference approximation to the Jacobian.

`NEQBJ` Solves a system of nonlinear equations using factored secant update with a user-supplied Jacobian.

`NEQNF` Solves a system of nonlinear equations using a modified Powell hybrid algorithm and a finite-difference approximation to the Jacobian.

`NEQNJ` Solves a system of nonlinear equations using a modified Powell hybrid algorithm with a user-supplied Jacobian.

G........... OPTIMIZATION (*search also classes K, L8*)

G1......... Unconstrained

G1a....... Univariate

G1a1..... Smooth function

G1a1a... User provides no derivatives

`UVMIF` Finds the minimum point of a smooth function of a single variable using only function evaluations.

G1a1b... User provides first derivatives

`UVMID` Finds the minimum point of a smooth function of a single variable using both function evaluations and first derivative evaluations.

G1a2..... General function (no smoothness assumed)

`UVMGS` Finds the minimum point of a nonsmooth function of a single variable.

G1b....... Multivariate

G1b1..... Smooth function

G1b1a... User provides no derivatives

`UMCGF` Minimizes a function of `N` variables using a conjugate gradient algorithm and a finite-difference gradient.

`UMINF` Minimizes a function of `N` variables using a quasi-New method and a finite-difference gradient.

`UNLSF` Solves a nonlinear least squares problem using a modified Levenberg-Marquardt algorithm and a finite-difference Jacobian.

G1b1b... User provides first derivatives

`UMCGG` Minimizes a function of `N` variables using a conjugate gradient algorithm and a user-supplied gradient.

`UMIDH` Minimizes a function of `N` variables using a modified Newton method and a finite-difference Hessian.

`UMING` Minimizes a function of `N` variables using a quasi-New method and a user-supplied gradient.

`UNLSJ` Solves a nonlinear least squares problem using a modified Levenberg-Marquardt algorithm and a user-supplied Jacobian.

G1b1c...User provides first and second derivatives

`UMIAH` Minimizes a function of `N` variables using a modified Newton method and a user-supplied Hessian.

G1b2.....General function (no smoothness assumed)

`UMPOL` Minimizes a function of `N` variables using a direct search polytope algorithm.

G2.........Constrained

G2a.......Linear programming

G2a1.....Dense matrix of constraints

`DLPRS` Solves a linear programming problem via the revised simplex algorithm.

G2a2.....Sparse matrix of constraints

`SLPRS` Solves a sparse linear programming problem via the revised simplex algorithm.

G2e.......Quadratic programming

G2e1.....Positive definite Hessian (i.e., convex problem)

`QPROG` Solves a quadratic programming problem subject to linear equality/inequality constraints.

G2h.......General nonlinear programming

G2h1.....Simple bounds

G2h1a...Smooth function

G2h1a1.User provides no derivatives

`BCLSF` Solves a nonlinear least squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm and a finite-difference Jacobian.

`BCONF` Minimizes a function of `N` variables subject to bounds the variables using a quasi-Newton method and a finite-difference gradient.

G2h1a2.User provides first derivatives

`BCLSJ` Solves a nonlinear least squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm and a user-supplied Jacobian.

BCODH Minimizes a function of N variables subject to bounds the variables using a modified Newton method and a finite-difference Hessian.

BCONG Minimizes a function of N variables subject to bounds the variables using a quasi-Newton method and a user-supplied gradient.

G2h1a3. User provides first and second derivatives

BCOAH Minimizes a function of N variables subject to bounds the variables using a modified Newton method and a user-supplied Hessian.

G2h1b... General function (no smoothness assumed)

BCPOL Minimizes a function of N variables subject to bounds the variables using a direct search complex algorithm.

G2h2..... Linear equality or inequality constraints

G2h2a... Smooth function

G2h2a1. User provides no derivatives

LCONF Minimizes a general objective function subject to linear equality/inequality constraints.

G2h2a2. User provides first derivatives

LCONG Minimizes a general objective function subject to linear equality/inequality constraints.

G2h3..... Nonlinear constraints

G2h3b... Equality and inequality constraints

NNLPG Uses a sequential equality constrained QP method.

NNLPF Uses a sequential equality constrained QP method.

G2h3b1. Smooth function and constraints

G2h3b1a. User provides no derivatives

G2h3b1b User provides first derivatives of function and constraints

G4......... Service routines

G4c....... Check user-supplied derivatives

CHGRD Checks a user-supplied gradient of a function.

CHHES Checks a user-supplied Hessian of an analytic function.

CHJAC Checks a user-supplied Jacobian of a system of equations with M functions in N unknowns.

G4d....... Find feasible point

GGUES Generates points in an N-dimensional space.

G4f....... Other

CDGRD Approximates the gradient using central differences.

FDGRD Approximates the gradient using forward differences.

`FDHES` Approximates the Hessian using forward differences and function values.

`FDJAC` Approximates the Jacobian of `M` functions in `N` unknowns using forward differences.

`GDHES` Approximates the Hessian using forward differences and a user-supplied gradient.

H...........DIFFERENTIATION, INTEGRATION

H1.........Numerical differentiation

`DERIV` Computes the first, second or third derivative of a user-supplied function.

H2.........Quadrature (numerical evaluation of definite integrals)

H2a........One-dimensional integrals

H2a1.....Finite interval (general integrand)

H2a1a...Integrand available via user-defined procedure

H2a1a1. Automatic (user need only specify required accuracy)

`QDAG` Integrates a function using a globally adaptive scheme based on Gauss-Kronrod rules.

`QDAGS` Integrates a function (which may have endpoint singularities).

`QDNG` Integrates a smooth function using a nonadaptive rule.

H2a2.....Finite interval (specific or special type integrand including weight functions, oscillating and singular integrands, principal value integrals, splines, etc.)

H2a2a...Integrand available via user-defined procedure

H2a2a1.Automatic (user need only specify required accuracy)

`QDAGP` Integrates a function with singularity points given.

`QDAWC` Integrates a function `F`(`X`)/(`X` – `C`) in the Cauchy principal value sense.

`QDAWO` Integrates a function containing a sine or a cosine.

`QDAWS` Integrates a function with algebraic-logarithmic singularities.

H2a2b...Integrand available only on grid

H2a2b1.Automatic (user need only specify required accuracy)

`BSITG` Evaluates the integral of a spline, given its B-spline representation.

H2a3.....Semi-infinite interval (including $e–^{x}$ weight function)

H2a3a...Integrand available via user-defined procedure

H2a3a1. Automatic (user need only specify required accuracy)

`QDAGI` Integrates a function over an infinite or semi-infinite interval.

`QDAWF` Computes a Fourier integral.

H2b....... Multidimensional integrals

H2b1..... One or more hyper-rectangular regions (including iterated integrals)

`QMC` Integrates a function over a hyperrectangle using a quasi-Monte Carlo method.

H2b1a... Integrand available via user-defined procedure

H2b1a1. Automatic (user need only specify required accuracy)

`QAND` Integrates a function on a hyper-rectangle.

`TWODQ` Computes a two-dimensional iterated integral.

H2b1b... Integrand available only on grid

H2b1b2. Nonautomatic

`BS2IG` Evaluates the integral of a tensor-product spline on a rectangular domain, given its tensor-product B-spline representation.

`BS3IG` Evaluates the integral of a tensor-product spline in three dimensions over a three-dimensional rectangle, given its tensorproduct B-spline representation.

H2c....... Service routines (compute weight and nodes for quadrature formulas)

`FQRUL` Computes a Fejér quadrature rule with various classical weight functions.

`GQRCF` Computes a Gauss, Gauss-Radau or Gauss-Lobatto quadrature rule given the recurrence coefficients for the monic polynomials orthogonal with respect to the weight function.

`GQRUL` Computes a Gauss, Gauss-Radau, or Gauss-Lobatto quadrature rule with various classical weight functions.

`RECCF` Computes recurrence coefficients for various monic polynomials.

`RECQR` Computes recurrence coefficients for monic polynomials given a quadrature rule.

I .... DIFFERENTIAL AND INTEGRAL EQUATIONS

I1 .... Ordinary differential equations (ODE's)

I1a. .... Initial value problems

I1a1 ...... General, nonstiff or mildly stiff

I1a1a..... One-step methods (e.g., Runge-Kutta)

`IVMRK` Solves an initial-value problem $y' = f(t, y)$ for ordinary differential equations using Runge-Kutta pairs of various orders.

`IVPRK` Solves an initial-value problem for ordinary differential equations using the Runge-Kutta-Verner fifth-order and sixth-order method.

I1a1b. ... Multistep methods (e.g., Adams predictor-corrector)

`IVPAG` Solves an initial-value problem for ordinary differential equations using either Adams-Moulton's or Gear's BDF method.

I1a2 ......Stiff and mixed algebraic-differential equations

`DASPG` Solves a first order differential-algebraic system of equations, $g(t, y, y') = 0$, using Petzold–Gear BDF method.

I1b ....Multipoint boundary value problems

I1b2 ......Nonlinear

`BVPFD` Solves a (parameterized) system of differential equations with boundary conditions at two points, using a variable order, variable step size finite-difference method with deferred corrections.

`BVPMS` Solves a (parameterized) system of differential equations with boundary conditions at two points, using a multiple-shooting method.

I1b3 ......Eigenvalue (e.g., Sturm-Liouville)

`SLCNT` Calculates the indices of eigenvalues of a Sturm-Liouville problem with boundary conditions (at regular points) in a specified subinterval of the real line, $[\alpha, \beta]$.

`SLEIG` Determines eigenvalues, eigenfunctions and/or spectral density functions for Sturm-Liouville problems in the form with boundary conditions (at regular points).

I2 ....Partial differential equations

I2a. ....Initial boundary value problems

I2a1 ......Parabolic

`PDE_1D_MG` Integrates an initial-value PDE problem with one space variable.

I2a1a.....One spatial dimension

`MOLCH` Solves a system of partial differential equations of the form $u_t = f(x, t, u, u_x, u_{xx})$ using the method of lines. The solution is represented with cubic Hermite polynomials.

I2b ....Elliptic boundary value problems

I2b1 ......Linear

I2b1a. ...Second order

I2b1a1...Poisson (Laplace) or Helmholtz equation

I2b1a1a.Rectangular domain (or topologically rectangular in the coordinate system)

`FPS2H` Solves Poisson's or Helmholtz's equation on a two-dimensional rectangle using a fast Poisson solver based on the `HODIE` finite-difference scheme on a uni mesh.

FPS3H Solves Poisson's or Helmholtz's equation on a three-dimensional box using a fast Poisson solver based on the HODIE finite-difference scheme on a uniform mesh.

J............INTEGRAL TRANSFORMS

J1..........Trigonometric transforms including fast Fourier transforms

J1a........One-dimensional

J1a1......Real

FFTRB Computes the real periodic sequence from its Fourier coefficients.

FFTRF Computes the Fourier coefficients of a real periodic sequence.

FFTRI Computes parameters needed by FFTRF and FFTRB.

J1a2......Complex

FAST-DFT Computes the Discrete Fourier Transform (DFT) of a rank-1 complex array, *x*.

FFTCB Computes the complex periodic sequence from its Fourier coefficients.

FFTCF Computes the Fourier coefficients of a complex periodic sequence.

FFTCI Computes parameters needed by FFTCF and FFTCB.

J1a3......Sine and cosine transforms

FCOSI Computes parameters needed by FCOST.

FCOST Computes the discrete Fourier cosine transformation of an even sequence.

FSINI Computes parameters needed by FSINT.

FSINT Computes the discrete Fourier sine transformation of an odd sequence.

QCOSB Computes a sequence from its cosine Fourier coefficients with only odd wave numbers.

QCOSF Computes the coefficients of the cosine Fourier transform with only odd wave numbers.

QCOSI Computes parameters needed by QCOSF and QCOSB.

QSINB Computes a sequence from its sine Fourier coefficients with only odd wave numbers.

QSINF Computes the coefficients of the sine Fourier transform with only odd wave numbers.

QSINI Computes parameters needed by QSINF and QSINB.

J1b........Multidimensional

FFT2B Computes the inverse Fourier transform of a complex periodic two-dimensional array.

FFT2D Computes Fourier coefficients of a complex periodic two-dimensional array.

FFT3B Computes the inverse Fourier transform of a complex periodic three-dimensional array.

FFT3F Computes Fourier coefficients of a complex periodic threedimensional array.

FAST_2DFT Computes the Discrete Fourier Transform (DFT) of a rank-2 complex array, *x*.

FAST_3DFT Computes the Discrete Fourier Transform (DFT) of a rank-3 complex array, *x*.

J2..........Convolutions

CCONV Computes the convolution of two complex vectors.

RCONV Computes the convolution of two real vectors.

J3..........Laplace transforms

INLAP Computes the inverse Laplace transform of a complex function.

SINLP Computes the inverse Laplace transform of a complex function.

K...........APPROXIMATION (*search also class L8*)

K1.........Least squares (L_2) approximation

K1a.......Linear least squares (*search also classes D5, D6, D9*)

K1a1.....Unconstrained

K1a1a...Univariate data (curve fitting)

K1a1a1 .Polynomial splines (piecewise polynomials)

BSLSQ Computes the least-squares spline approximation, and return the B-spline coefficients.

BSVLS Computes the variable knot B-spline least squares approximation to given data.

CONFT Computes the least-squares constrained spline approximation, returning the B-spline coefficients.

FRENCH_CURVE Constrained weighted least-squares fitting of B-splines to discrete data, with covariance matrix.and constraints at points.

K1a1a2 .Polynomials

RCURV Fits a polynomial curve using least squares.

K1a1a3 .Other functions (e.g., trigonometric, user-specified)

FNLSQ Compute a least-squares approximation with user-supplied basis functions.

K1a1b...Multivariate data (surface fitting)

BSLS2 Computes a two-dimensional tensor-product spline approximant using least squares, returning the tensor-product B-spline coefficients.

`BSLS3` Computes a three-dimensional tensor-product spline approximant using least squares, returning the tensor-product B-spline coefficients.

`SURFACE_FAIRING` Constrained weighted least-squares fitting of tensor product B-splines to discrete data, with covariance matrix and constraints at points.

K1a2.....Constrained

`LIN_SOL_LSQ_CON` Routine for constrained linear-least squares based on a least-distance, dual algorithm.

`LIN_SOL_LSQ_INQ` Routine for constrained linear-least squares based on a least-distance, dual algorithm.

`LEAST_PROJ_`
`DISTANCE` Routine for constrained linear-least squares based on a least-distance, dual algorithm.

`PARALLEL_&`
`NONONEGATIVE_LSQ` Solves multiple systems of linear equations $A_j x_j = y_j, j = 1, \ldots, k$. Each matrix A_j is tridiagonal with the same dimension, n: The default solution method is based on LU factorization computed using cyclic reduction. An option is used to select Gaussian elimination with partial pivoting.

`PARALLEL_& BOUNDED_LSQ`
Parallel routines for simple bounded constrained linear-least squares based on a descent algorithm.

K1a2a...Linear constraints

`LCLSQ` Solves a linear least-squares problem with linear constraints.

`PARALLEL_`
`NONNEGATIVE_LSQ` Solves a large least-squares system with non-negative constraints, using parallel computing.

`PARALLEL_`
`BOUNDED_LSQ` Solves a large least-squares system with simple bounds, using parallel computing.

K1b.......Nonlinear least squares

K1b1.....Unconstrained

K1b1a...Smooth functions

K1b1a1.User provides no derivatives

`UNLSF` Solves a nonlinear least squares problem using a modified Levenberg-Marquardt algorithm and a finite-difference Jacobian.

K1b1a2.User provides first derivatives

`UNLSJ` Solves a nonlinear least squares problem using a modified Levenberg-Marquardt algorithm and a user-supplied Jacobian.

K1b2.....Constrained

K1b2a...Linear constraints

BCLSF Solves a nonlinear least squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm and a finite-difference Jacobian.

BCLSJ Solves a nonlinear least squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm and a user-supplied Jacobian.

BCNLS Solves a nonlinear least-squares problem subject to bounds on the variables and general linear constraints.

K2.........Minimax (L_∞) approximation

RATCH Computes a rational weighted Chebyshev approximation to a continuous function on an interval.

K5.........Smoothing

CSSCV Computes a smooth cubic spline approximation to noisy data using cross-validation to estimate the smoothing parameter.

CSSED Smooths one-dimensional data by error detection.

CSSMH Computes a smooth cubic spline approximation to noisy data.

K6.........Service routines for approximation

K6a........Evaluation of fitted functions, including quadrature

K6a1.....Function evaluation

BSVAL Evaluates a spline, given its B-spline representation.

CSVAL Evaluates a cubic spline.

PPVAL Evaluates a piecewise polynomial.

K6a2.....Derivative evaluation

BSDER Evaluates the derivative of a spline, given its B-spline representation.

CS1GD Evaluates the derivative of a cubic spline on a grid.

CSDER Evaluates the derivative of a cubic spline.

PP1GD Evaluates the derivative of a piecewise polynomial on a grid.

PPDER Evaluates the derivative of a piecewise polynomial.

K6a3.....Quadrature

CSITG Evaluates the integral of a cubic spline.

PPITG Evaluates the integral of a piecewise polynomial.

K6c........Manipulation of basis functions (e.g., evaluation, change of basis)

BSCPP Converts a spline in B-spline representation to piecewise polynomial representation.

L...........STATISTICS, PROBABILITY

L1.........Data summarization

L1c. ......Multi-dimensional data

L1c1 ..... Raw data

L1c1b. .. Covariance, correlation

CCORL Computes the correlation of two complex vectors.
RCORL Computes the correlation of two real vectors.

L3 .... Elementary statistical graphics (*search also class Q*)

L3e. ...... Multi-dimensional data

L3e3. .... Scatter diagrams

L3e3a. .. Superimposed *Y* vs. *X*

PLOTP Prints a plot of up to 10 sets of points.

L6 .... Random number generation

L6a. ...... Univariate

RAND_GEN Generates a rank-1 array of random numbers. The output array entries are positive and less than 1 in value.

L6a21 ... Uniform (continuous, discrete), uniform order statistics

RNUN Generates pseudorandom numbers from a uniform (0, 1) distribution.

RNUNF Generates a pseudorandom number from a uniform (0, 1) distribution.

L6b .... Mulitivariate

L6b21 ... Linear L-1 (least absolute value) approximation random numbers

FAURE_INIT Shuffles Faure sequence initialization.

FAURE_FREE Frees the structure containing information about the Faure sequence.

FAURE_NEXT Computes a shuffled Faure sequence.

L6c. ...... Service routines (e.g., seed)

RNGET Retrieves the current value of the seed used in the IMSL random number generators.

RNOPT Selects the uniform (0, 1) multiplicative congruential pseudorandom number generator.

RNSET Initializes a random seed for use in the IMSL random number generators.

RAND_GEN Generates a rank-1 array of random numbers. The output array entries are positive and less than 1 in value.

L8 .... Regression (*search also classes D5, D6, D9, G, K*)

L8a. ...... Simple linear (e.g., $y = \beta_0 + \beta_1 x + \varepsilon$) (*search also class L8h*)

L8a1. .... Ordinary least squares

FNLSQ Computes a least-squares approximation with user-supplied basis functions.

L8a1a ... Parameter estimation

L8a1a1. Unweighted data

RLINE Fits a line to a set of data points using least squares.

L8b. ......Polynomial (e.g., y = $\beta_0 + \beta_1 x + \beta_2 x2 + \varepsilon$) (*search also class L8c*)

L8b1 .....Ordinary least squares

L8b1b ...Parameter estimation

L8b1b2. Using orthogonal polynomials

RCURV Fits a polynomial curve using least squares.

L8c ....Multiple linear (e.g., $y = \beta_0 + \beta_1 x_1 + \ldots + \beta_k x_k + \varepsilon$)

L8c1 .....Ordinary least squares

L8c1b ...Parameter estimation (*search also class L8c1a*)

L8c1b1 .Using raw data

LSBRR Solves a linear least-squares problem with iterative refinement.

LSQRR Solves a linear least-squares problem without iterative refinement.

N...........DATA HANDLING

N1.........Input, output

PGOPT Sets or retrieves page width and length for printing.

WRCRL Prints a complex rectangular matrix with a given format and labels.

WRCRN Prints a complex rectangular matrix with integer row and column labels.

WRIRL Prints an integer rectangular matrix with a given format and labels.

WRIRN Prints an integer rectangular matrix with integer row and column labels.

WROPT Sets or retrieves an option for printing a matrix.

WRRRL Prints a real rectangular matrix with a given format and labels.

WRRRN Prints a real rectangular matrix with integer row and column labels.

SCALAPACK_READ Reads matrix data from a file and place in a two-dimensional block-cyclic form on a process grid.

SCALAPACK_WRITE Writes matrix data to a file, starting with a two-dimensional block-cyclic form on a process grid.

SHOW Prints rank-1 and rank-2 arrays with indexing and text.

N3.........Character manipulation

ACHAR Returns a character given its ASCII value.

CVTSI Converts a character string containing an integer number into the corresponding integer form.

IACHAR Returns the integer ASCII value of a character argument.

ICASE Returns the ASCII value of a character converted to uppercase.

IICSR Compares two character strings using the ASCII collating sequence but without regard to case.

IIDEX Determines the position in a string at which a given character sequence begins without regard to case.

N4......... Storage management (e.g., stacks, heaps, trees)

IWKCIN Initializes bookkeeping locations describing the character workspace stack.

IWKIN Initializes bookkeeping locations describing the workspace stack.

ScaLAPACK_READ Moves data from a file to Block-Cyclic form, for use in ScaLAPACK.

ScaLAPACK_WRITE Move data from Block-Cyclic form, following use in ScaLAPACK, to a file.

N5......... Searching

N5b....... Insertion position

ISRCH Searches a sorted integer vector for a given integer and return its index.

SRCH Searches a sorted vector for a given scalar and return its index.

SSRCH Searches a character vector, sorted in ascending ASCII order, for a given string and return its index.

N5c....... On a key

IIDEX Determines the position in a string at which a given character sequence begins without regard to case.

ISRCH Searches a sorted integer vector for a given integer and return its index.

SRCH Searches a sorted vector for a given scalar and return its index.

SSRCH Searches a character vector, sorted in ascending ASCII order, for a given string and return its index.

N6......... Sorting

N6a....... Internal

N6a1..... Passive (i.e., construct pointer array, rank)

N6a1a... Integer

SVIBP Sorts an integer array by nondecreasing absolute value and return the permutation that rearranges the array.

SVIGP Sorts an integer array by algebraically increasing value and return the permutation that rearranges the array.

N6a1b... Real

SVRBP Sorts a real array by nondecreasing absolute value and return the permutation that rearranges the array.

SVRGP Sorts a real array by algebraically increasing value and return the permutation that rearranges the array.

LIN_SOL_TRI Sorts a rank-1 array of real numbers *x* so the *y* results are algebraically nondecreasing, $y_1 \le y_2 \le \ldots y_n$.

N6a2.....Active

N6a2a...Integer

SVIBN Sorts an integer array by nondecreasing absolute value.

SVIBP Sorts an integer array by nondecreasing absolute value and return the permutation that rearranges the array.

SVIGN Sorts an integer array by algebraically increasing value.

SVIGP Sorts an integer array by algebraically increasing value and return the permutation that rearranges the array.

N6a2b...Real

SVRBN Sorts a real array by nondecreasing absolute value.

SVRBP Sorts a real array by nondecreasing absolute value and return the permutation that rearranges the array.

SVRGN Sorts a real array by algebraically increasing value.

SVRGP Sorts a real array by algebraically increasing value and return the permutation that rearranges the array.

N8.........Permuting

PERMA Permutes the rows or columns of a matrix.

PERMU Rearranges the elements of an array as specified by a permutation.

Q...........GRAPHICS (*search also classes L3*)

PLOTP Prints a plot of up to 10 sets of points.

R...........SERVICE ROUTINES

IDYWK Computes the day of the week for a given date.

IUMAG Sets or retrieves MATH/LIBRARY integer options.

NDAYS Computes the number of days from January 1, 1900, to the given date.

NDYIN Gives the date corresponding to the number of days since January 1, 1900.

SUMAG Sets or retrieves MATH/LIBRARY single-precision options.

TDATE Get stoday's date.

TIMDY Gets time of day.

VERML Obtains IMSL MATH/LIBRARY-related version, system and license numbers.

R1.........Machine-dependent constants

AMACH Retrieves single-precision machine constants.

IFNAN Checks if a value is NaN (not a number).

IMACH Retrieves integer machine constants.

ISNAN Detects an IEEE NaN (not-a-number).

NAN Returns, as a scalar function, a value corresponding to the IEEE 754 Standard format of floating point (ANSI/IEEE 1985) for NaN.

UMACH Sets or retrieves input or output device unit numbers.

R3......... Error handling

BUILD_ERROR _STRUCTURE	Fills in flags, values and update the data structure for error conditions that occur in Library routines. Prepares the structure so that calls to routine `error_post` will display the reason for the error.

R3b....... Set unit number for error messages

UMACH	Sets or retrieves input or output device unit numbers.

R3c....... Other utilities

ERROR_POST	Prints error messages that are generated by IMSL Library routines.
ERSET	Sets error handler default print and stop actions.
IERCD	Retrieves the code for an informational error.
N1RTY	Retrieves an error type for the most recently called IMSL routine.

S. .... SOFTWARE DEVELOPMENT TOOLS

S3 .... Dynamic program analysis tools

CPSEC	Returns CPU time used in seconds.

Appendix B: Alphabetical Summary of Routines

IMSL MATH/LIBRARY

ACBCB	1441	Adds two complex band matrices, both in band storage mode.
ACHAR	1624	Returns a character given its ASCII value.
AMACH	1685	Retrieves single-precision machine constants.
ARBRB	1438	Adds two band matrices, both in band storage mode.
BCLSF	1274	Solves a nonlinear least squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm and a finite-difference Jacobian.
BCLSJ	1281	Solves a nonlinear least squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm and a user-supplied Jacobian.
BCNLS	1288	Solves a nonlinear least-squares problem subject to bounds on the variables and general linear constraints.
BCOAH	1263	Minimizes a function of `N` variables subject to bounds the variables using a modified Newton method and a user-supplied Hessian.
BCODH	1257	Minimizes a function of `N` variables subject to bounds the variables using a modified Newton method and a finite-difference Hessian.
BCONF	1243	Minimizes a function of `N` variables subject to bounds the variables using a quasi-Newton method and a finite-difference gradient.
BCONG	1249	Minimizes a function of `N` variables subject to bounds the variables using a quasi-Newton method and a user-supplied gradient.
BCPOL	1271	Minimizes a function of `N` variables subject to bounds the variables using a direct search complex algorithm.

BLINF	1427	Computes the bilinear form x^TAy.
BS1GD	656	Evaluates the derivative of a spline on a grid, given its B-spline representation.
BS2DR	653	Evaluates the derivative of a two-dimensional tensor-product spline, given its tensor-product B-spline representation.
BS2GD	656	Evaluates the derivative of a two-dimensional tensor-product spline, given its tensor-product B-spline representation on a grid.
BS2IG	661	Evaluates the integral of a tensor-product spline on a rectangular domain, given its tensor-product B-spline representation.
BS2IN	631	Computes a two-dimensional tensor-product spline interpolant, returning the tensor-product B-spline coefficients.
BS2VL	651	Evaluates a two-dimensional tensor-product spline, given its tensor-product B-spline representation.
BS3DR	666	Evaluates the derivative of a three-dimensional tensor-product spline, given its tensor-product B-spline representation.
BS3GD	670	Evaluates the derivative of a three-dimensional tensor-product spline, given its tensor-product B-spline representation on a grid.
BS3IG	676	Evaluates the integral of a tensor-product spline in three dimensions over a three-dimensional rectangle, given its tensorproduct B-spline representation.
BS3IN	635	Computes a three-dimensional tensor-product spline interpolant, returning the tensor-product B-spline coefficients.
BS3VL	664	Evaluates a three-dimensional tensor-product spline, given its tensor-product B-spline representation.
BSCPP	680	Converts a spline in B-spline representation to piecewise polynomial representation.
BSDER	643	Evaluates the derivative of a spline, given its B-spline representation.
BSINT	622	Computes the spline interpolant, returning the B-spline coefficients.
BSITG	649	Evaluates the integral of a spline, given its B-spline representation.

BSLS2	743	Computes a two-dimensional tensor-product spline approximant using least squares, returning the tensor-product B-spline coefficients.
BSLS3	748	Computes a three-dimensional tensor-product spline approximant using least squares, returning the tensor-product B-spline coefficients.
BSLSQ	725	Computes the least-squares spline approximation, and return the B-spline coefficients.
BSNAK	625	Computes the ‘not-a-knot’ spline knot sequence.
BSOPK	628	Computes the ‘optimal’ spline knot sequence.
BSVAL	641	Evaluates a spline, given its B-spline representation.
BSVLS	729	Computes the variable knot B-spline least squares approximation to given data.
BVPFD	870	Solves a (parameterized) system of differential equations with boundary conditions at two points, using a variable order, variable step size finite-difference method with deferred corrections.
BVPMS	882	Solves a (parameterized) system of differential equations with boundary conditions at two points, using a multiple-shooting method.
CADD	1319	Adds a scalar to each component of a vector, $x \leftarrow x + a$, all complex.
CAXPY	1320	Computes the scalar times a vector plus a vector, $y \leftarrow ax + y$, all complex.
CCBCB	1393	Copies a complex band matrix stored in complex band storage mode.
CCBCG	1400	Converts a complex matrix in band storage mode to a complex matrix in full storage mode.
CCGCB	1398	Converts a complex general matrix to a matrix in complex band storage mode.
CCGCG	1390	Copies a complex general matrix.
CCONV	1064	Computes the convolution of two complex vectors.
CCOPY	1319	Copies a vector x to a vector y, both complex.
CCORL	1073	Computes the correlation of two complex vectors.
CDGRD	1336	Approximates the gradient using central differences.
CDOTC	1320	Computes the complex conjugate dot product, $\bar{x}^T y$.
CDOTU	1320	Computes the complex dot product $x^T y$.

CGBMV 1330 Computes one of the matrix-vector operations:
$y \leftarrow \alpha Ax + \beta y,\ y \leftarrow \alpha A^T x + \beta y,$ or $y \leftarrow \alpha \overline{A}^T + \beta y$,
where A is a matrix stored in band storage mode.

CGEMM 1333 Computes one of the matrix-matrix operations:
$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha A^T B + \beta C, C \leftarrow \alpha AB^T$
$+\beta C, C \leftarrow \alpha A^T B^T + \beta C, C \leftarrow \alpha A\overline{B}^T + \beta C,$
or $C \leftarrow \alpha \overline{A}^T B + \beta C, C \leftarrow \alpha A^T \overline{B}^T + \beta C,$
$C \leftarrow \alpha \overline{A}^T B^T + \beta C,$ or $C \leftarrow \alpha \overline{A}^T \overline{B}^T + \beta C$

CGEMV 1329 Computes one of the matrix-vector operations:
$y \leftarrow \alpha Ax + \beta y,\ y \leftarrow \alpha A^T x + \beta y,$ or $y \leftarrow \alpha \overline{A}^T + \beta y$,

CGERC 1384 Computes the rank-one update of a complex general matrix:
$A \leftarrow A + \alpha x\overline{y}^T$.

CGERU 1384 Computes the rank-one update of a complex general matrix:
$A \leftarrow A + \alpha xy^T$.

CHBCB 1411 Copies a complex Hermitian band matrix stored in band Hermitian storage mode to a complex band matrix stored in band storage mode.

CHBMV 1381 Computes the matrix-vector operation
$y \leftarrow \alpha Ax + \beta y$,
where A is an Hermitian band matrix in band Hermitian storage.

CHEMM 1385 Computes one of the matrix-matrix operations:
$C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$,
where A is an Hermitian matrix and B and C are m by n matrices.

CHEMV 1381 Computes the matrix-vector operation
$y \leftarrow \alpha Ax + \beta y$,
where A is an Hermitian matrix.

CHER 1384 Computes the rank-one update of an Hermitian matrix:
$A \leftarrow A + \alpha x\overline{x}^T$ with x complex and α real.

CHER2 1384 Computes a rank-two update of an Hermitian matrix:
$A \leftarrow A + \alpha x\overline{y}^T + \overline{\alpha} y\overline{x}^T$.

CHER2K 1387 Computes one of the Hermitian rank $2k$ operations:
$C \leftarrow \alpha A\overline{B}^T + \overline{\alpha} B\overline{A}^T + \beta C$ or $C \leftarrow \alpha \overline{A}^T B + \overline{\alpha}\overline{B}^T A + \beta C$,
where C is an n by n Hermitian matrix and A and B are n

		by *k* matrices in the first case and *k* by *n* matrices in the second case.
CHERK	1386	Computes one of the Hermitian rank *k* operations: $C \leftarrow \alpha A\overline{A}^T + \beta C$ or $C \leftarrow \alpha\overline{A}^T A + \beta C$, where *C* is an *n* by *n* Hermitian matrix and *A* is an *n* by *k* matrix in the first case and a *k* by *n* matrix in the second case.
CHFCG	1408	Extends a complex Hermitian matrix defined in its upper triangle to its lower triangle.
CHGRD	1349	Checks a user-supplied gradient of a function.
CHHES	1352	Checks a user-supplied Hessian of an analytic function.
CHJAC	1355	Checks a user-supplied Jacobian of a system of equations with `M` functions in `N` unknowns.
CHOL	1475	Computes the Cholesky factorization of a positive-definite, symmetric or self-adjoint matrix, *A*.
COND	1476	Computes the condition number of a rectangular matrix, *A*.
CONFT	734	Computes the least-squares constrained spline approximation, returning the B-spline coefficients.
CONST	1669	Returns the value of various mathematical and physical constants.
CPSEC	1631	Returns CPU time used in seconds.
CRBCB	1405	Converts a real matrix in band storage mode to a complex matrix in band storage mode.
CRBRB	1392	Copies a real band matrix stored in band storage mode.
CRBRG	1397	Converts a real matrix in band storage mode to a real general matrix.
CRGCG	1402	Copies a real general matrix to a complex general matrix.
CRGRB	1395	Converts a real general matrix to a matrix in band storage mode.
CRGRG	1389	Copies a real general matrix.
CRRCR	1403	Copies a real rectangular matrix to a complex rectangular matrix.
CS1GD	602	Evaluates the derivative of a cubic spline on a grid.
CSAKM	500	Computes the Akima cubic spline interpolant.
CSBRB	1409	Copies a real symmetric band matrix stored in band symmetric storage mode to a real band matrix stored in band storage mode.

CSCAL	1319	Multiplies a vector by a scalar, $y \leftarrow ay$, both complex.
CSCON	603	Computes a cubic spline interpolant that is consistent with the concavity of the data.
CSDEC	593	Computes the cubic spline interpolant with specified derivative endpoint conditions.
CSDER	610	Evaluates the derivative of a cubic spline.
CSET	1318	Sets the components of a vector to a scalar, all complex.
CSFRG	1406	Extends a real symmetric matrix defined in its upper triangle to its lower triangle.
CSHER	597	Computes the Hermite cubic spline interpolant.
CSIEZ	587	Computes the cubic spline interpolant with the 'not-a-knot' condition and return values of the interpolant at specified points.
CSINT	590	Computes the cubic spline interpolant with the 'not-a-knot' condition.
CSITG	616	Evaluates the integral of a cubic spline.
CSPER	506	Computes the cubic spline interpolant with periodic boundary conditions.
CSROT	1325	Applies a complex Givens plane rotation.
CSROTM	1326	Applies a complex modified Givens plane rotation.
CSSCAL	1319	Multiplies a complex vector by a single-precision scalar, $y \leftarrow ay$.
CSSCV	761	Computes a smooth cubic spline approximation to noisy data using cross-validation to estimate the smoothing parameter.
CSSED	754	Smooths one-dimensional data by error detection.
CSSMH	758	Computes a smooth cubic spline approximation to noisy data.
CSUB	1319	Subtracts each component of a vector from a scalar, $x \leftarrow a - x$, all complex.
CSVAL	609	Evaluates a cubic spline.
CSVCAL	1319	Multiplies a complex vector by a single-precision scalar and store the result in another complex vector, $y \leftarrow ax$.
CSWAP	1320	Interchanges vectors x and y, both complex.
CSYMM	1334	Computes one of the matrix-matrix operations: $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$, where A is a symmetric matrix and B and C are m by n matrices.

CSYR2K	1335	Computes one of the symmetric rank 2*k* operations: $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$ or $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$, where *C* is an *n* by *n* symmetric matrix and *A* and *B* are *n* by *k* matrices in the first case and *k* by *n* matrices in the second case.
CSYRK	1334	Computes one of the symmetric rank *k* operations: $C \leftarrow \alpha AA^T + \beta C$ or $C \leftarrow \alpha A^T A + \beta C$, where *C* is an *n* by *n* symmetric matrix and *A* is an *n* by *k* matrix in the first case and a *k* by *n* matrix in the second case.
CTBMV	1331	Computes one of the matrix-vector operations: $x \leftarrow Ax, x \leftarrow A^T x,$ or $x \leftarrow \overline{A}^T x$, where *A* is a triangular matrix in band storage mode.
CTBSV	1332	Solves one of the complex triangular systems: $x \leftarrow A^{-1}x$, $x \leftarrow \left(A^{-1}\right)^T x,$ or $x \leftarrow \left(\overline{A}^T\right)^{-1} x$, where *A* is a triangular matrix in band storage mode.
CTRMM	1335	Computes one of the matrix-matrix operations: $B \leftarrow \alpha AB, B \leftarrow \alpha A^T B, B \leftarrow \alpha BA, B \leftarrow \alpha BA^T$, $B \leftarrow \alpha \overline{A}^T B,$ or $B \leftarrow \alpha B\overline{A}^T$ where *B* is an *m* by *n* matrix and *A* is a triangular matrix.
CTRMV	1331	Computes one of the matrix-vector operations: $x \leftarrow Ax, x \leftarrow A^T x,$ or $x \leftarrow \overline{A}^T x$, where *A* is a triangular matrix.
CTRSM	1336	Solves one of the complex matrix equations: $B \leftarrow \alpha A^{-1}B, B \leftarrow \alpha BA^{-1}, B \leftarrow \alpha\left(A^{-1}\right)^T B, B \leftarrow \alpha B\left(A^{-1}\right)^T$, $B \leftarrow \alpha\left(\overline{A}^T\right)^{-1} B,$ or $B \leftarrow \alpha B\left(\overline{A}^T\right)^{-1}$ where *A* is a traiangular matrix.
CTRSV	1331	Solves one of the complex triangular systems: $x \leftarrow A^{-1}x, x \leftarrow \left(A^{-1}\right)^T x,$ or $x \leftarrow \left(\overline{A}^T\right)^{-1} x$, where *A* is a triangular matrix.
CUNIT	1672	Converts `X` in units `XUNITS` to `Y` in units `YUNITS`.
CVCAL	1319	Multiplies a vector by a scalar and store the result in another vector, $y \leftarrow ax$, all complex.
CVTSI	1630	Converts a character string containing an integer number into the corresponding integer form.

CZCDOT	1321	Computes the sum of a complex scalar plus a complex conjugate dot product, $a+\bar{x}^T y$, using a double-precision accumulator.
CZDOTA	1321	Computes the sum of a complex scalar, a complex dot product and the double-complex accumulator, which is set to the result ACC ← ACC + $a + x^T y$.
CZDOTC	1320	Computes the complex conjugate dot product, $\bar{x}^T y$, using a double-precision accumulator.
CZDOTI	1321	Computes the sum of a complex scalar plus a complex dot product using a double-complex accumulator, which is set to the result ACC ← $a + x^T y$.
CZDOTU	1320	Computes the complex dot product $x^T y$ using a double-precision accumulator.
CZUDOT	1321	Computes the sum of a complex scalar plus a complex dot product, $a + x^T y$, using a double-precision accumulator.
DASPG	889	Solves a first order differential-algebraic system of equations, $g(t, y, y') = 0$, using Petzold–Gear BDF method.
DERIV	827	Computes the first, second or third derivative of a user-supplied function.
DET	1477	Computes the determinant of a rectangular matrix, *A*.
DIAG	1479	Constructs a square diagonal matrix from a rank-1 array or several diagonal matrices from a rank-2 array.
DIAGONALS	1479	Extracts a rank-1 array whose values are the diagonal terms of a rank-2 array argument.
DISL1	1452	Computes the 1-norm distance between two points.
DISL2	1450	Computes the Euclidean (2-norm) distance between two points.
DISLI	1454	Computes the infinity norm distance between two points.
DLPRS	1297	Solves a linear programming problem via the revised simplex algorithm.
DMACH	1686	See AMACH.
DQADD	1460	Adds a double-precision scalar to the accumulator in extended precision.
DQINI	1460	Initializes an extended-precision accumulator with a double-precision scalar.

DQMUL	1460	Multiplies double-precision scalars in extended precision.
DQSTO	1460	Stores a double-precision approximation to an extended-precision scalar.
DSDOT	1371	Computes the single-precision dot product $x^T y$ using a double precision accumulator.
DUMAG	1664	This routine handles MATH/LIBRARY and STAT/LIBRARY type DOUBLE PRECISION options.
EIG	1480	Computes the eigenvalue-eigenvector decomposition of an ordinary or generalized eigenvalue problem.
EPICG	467	Computes the performance index for a complex eigensystem.
EPIHF	518	Computes the performance index for a complex Hermitian eigensystem.
EPIRG	460	Computes the performance index for a real eigensystem.
EPISB	501	Computes the performance index for a real symmetric eigensystem in band symmetric storage mode.
EPISF	483	Computes the performance index for a real symmetric eigensystem.
ERROR_POST	1568	Prints error messages that are generated by IMSL routines using EPACK
ERSET	1679	Sets error handler default print and stop actions.
EVAHF	508	Computes the largest or smallest eigenvalues of a complex Hermitian matrix.
EVASB	490	Computes the largest or smallest eigenvalues of a real symmetric matrix in band symmetric storage mode.
EVASF	473	Computes the largest or smallest eigenvalues of a real symmetric matrix.
EVBHF	513	Computes the eigenvalues in a given range of a complex Hermitian matrix.
EVBSB	495	Computes the eigenvalues in a given interval of a real symmetric matrix stored in band symmetric storage mode.
EVBSF	478	Computes selected eigenvalues of a real symmetric matrix.
EVCCG	464	Computes all of the eigenvalues and eigenvectors of a complex matrix.
EVCCH	526	Computes all of the eigenvalues and eigenvectors of a complex upper Hessenberg matrix.

EVCHF	505	Computes all of the eigenvalues and eigenvectors of a complex Hermitian matrix.
EVCRG	457	Computes all of the eigenvalues and eigenvectors of a real matrix.
EVCRH	522	Computes all of the eigenvalues and eigenvectors of a real upper Hessenberg matrix.
EVCSB	487	Computes all of the eigenvalues and eigenvectors of a real symmetric matrix in band symmetric storage mode.
EVCSF	471	Computes all of the eigenvalues and eigenvectors of a real symmetric matrix.
EVEHF	510	Computes the largest or smallest eigenvalues and the corresponding eigenvectors of a complex Hermitian matrix.
EVESB	492	Computes the largest or smallest eigenvalues and the corresponding eigenvectors of a real symmetric matrix in band symmetric storage mode.
EVESF	475	Computes the largest or smallest eigenvalues and the corresponding eigenvectors of a real symmetric matrix.
EVFHF	515	Computes the eigenvalues in a given range and the corresponding eigenvectors of a complex Hermitian matrix.
EVFSB	498	Computes the eigenvalues in a given interval and the corresponding eigenvectors of a real symmetric matrix stored in band symmetric storage mode.
EVFSF	480	Computes selected eigenvalues and eigenvectors of a real symmetric matrix.
EVLCG	462	Computes all of the eigenvalues of a complex matrix.
EVLCH	525	Computes all of the eigenvalues of a complex upper Hessenberg matrix.
EVLHF	502	Computes all of the eigenvalues of a complex Hermitian matrix.
EVLRG	455	Computes all of the eigenvalues of a real matrix.
EVLRH	520	Computes all of the eigenvalues of a real upper Hessenberg matrix.
EVLSB	485	Computes all of the eigenvalues of a real symmetric matrix in band symmetric storage mode.
EVLSF	469	Computes all of the eigenvalues of a real symmetric matrix.
EYE	1481	Creates a rank-2 square array whose diagonals are all the value one.

FAURE_FREE	1655	Frees the structure containing information about the Faure sequence.
FAURE_INIT	1655	Shuffled Faure sequence initialization.
FAURE_NEXT	1656	Computes a shuffled Faure sequence.
FAST_DFT	992	Computes the Discrete Fourier Transform of a rank-1 complex array, *x*.
FAST_2DFT	1000	Computes the Discrete Fourier Transform (2DFT) of a rank-2 complex array, *x*.
FAST_3DFT	1006	Computes the Discrete Fourier Transform (2DFT) of a rank-3 complex array, *x*.
FCOSI	1030	Computes parameters needed by `FCOST`.
FCOST	1028	Computes the discrete Fourier cosine transformation of an even sequence.
FDGRD	1338	Approximates the gradient using forward differences.
FDHES	1340	Approximates the Hessian using forward differences and function values.
FDJAC	1346	Approximates the Jacobian of `M` functions in `N` unknowns using forward differences.
FFT	1482	The Discrete Fourier Transform of a complex sequence and its inverse transform.
FFT_BOX	1482	The Discrete Fourier Transform of several complex or real sequences.
FFT2B	1048	Computes the inverse Fourier transform of a complex periodic two-dimensional array.
FFT2D	1045	Computes Fourier coefficients of a complex periodic two-dimensional array.
FFT3B	1055	Computes the inverse Fourier transform of a complex periodic three-dimensional array.
FFT3F	1051	Computes Fourier coefficients of a complex periodic threedimensional array.
FFTCB	1019	Computes the complex periodic sequence from its Fourier coefficients.
FFTCF	1017	Computes the Fourier coefficients of a complex periodic sequence.
FFTCI	1022	Computes parameters needed by `FFTCF` and `FFTCB`.
FFTRB	1012	Computes the real periodic sequence from its Fourier coefficients.

FFTRF	1009	Computes the Fourier coefficients of a real periodic sequence.
FFTRI	1015	Computes parameters needed by `FFTRF` and `FFTRB`.
FNLSQ	720	Computes a least-squares approximation with user-supplied basis functions.
FPS2H	961	Solves Poisson's or Helmholtz's equation on a two-dimensional rectangle using a fast Poisson solver based on the `HODIE` finite-difference scheme on a uni mesh.
FPS3H	967	Solves Poisson's or Helmholtz's equation on a three-dimensional box using a fast Poisson solver based on the `HODIE` finite-difference scheme on a uniform mesh.
FQRUL	824	Computes a Fejér quadrature rule with various classical weight functions.
FSINI	1026	Computes parameters needed by `FSINT`.
FSINT	1024	Computes the discrete Fourier sine transformation of an odd sequence.
GDHES	1343	Approximates the Hessian using forward differences and a user-supplied gradient.
GGUES	1359	Generates points in an `N`-dimensional space.
GMRES	368	Uses restarted `GMRES` with reverse communication to generate an approximate solution of $Ax = b$.
GPICG	542	Computes the performance index for a generalized complex eigensystem $Az = \lambda Bz$.
GPIRG	535	Computes the performance index for a generalized real eigensystem $Az = \lambda Bz$.
GPISP	549	Computes the performance index for a generalized real symmetric eigensystem problem.
GQRCF	815	Computes a Gauss, Gauss-Radau or Gauss-Lobatto quadrature rule given the recurrence coefficients for the monic polynomials orthogonal with respect to the weight function.
GQRUL	811	Computes a Gauss, Gauss-Radau, or Gauss-Lobatto quadrature rule with various classical weight functions.
GVCCG	540	Computes all of the eigenvalues and eigenvectors of a generalized complex eigensystem $Az = \lambda Bz$.
GVCRG	531	Computes all of the eigenvalues and eigenvectors of a generalized real eigensystem $Az = \lambda Bz$.

GVCSP	547	Computes all of the eigenvalues and eigenvectors of the generalized real symmetric eigenvalue problem $Az = \lambda Bz$, with B symmetric positive definite.
GVLCG	537	Computes all of the eigenvalues of a generalized complex eigensystem $Az = \lambda Bz$.
GVLRG	529	Computes all of the eigenvalues of a generalized real eigensystem $Az = \lambda Bz$.
GVLSP	544	Computes all of the eigenvalues of the generalized real symmetric eigenvalue problem $Az = \lambda Bz$, with B symmetric positive definite.
HRRRR	1425	Computes the Hadamard product of two real rectangular matrices.
HYPOT	1675	Computes $\sqrt{a^2 + b^2}$ without underflow or overflow.
IACHAR	1625	Returns the integer ASCII value of a character argument.
IADD	1319	Adds a scalar to each component of a vector, $x \leftarrow x + a$, all integer.
ICAMAX	1324	Finds the smallest index of the component of a complex vector having maximum magnitude.
ICAMIN	1323	Finds the smallest index of the component of a complex vector having minimum magnitude.
ICASE	1626	Returns the ASCII value of a character converted to uppercase.
ICOPY	1319	Copies a vector x to a vector y, both integer.
IDYWK	1637	Computes the day of the week for a given date.
IERCD	1680	Retrieves the code for an informational error.
IFFT	1483	The inverse of the Discrete Fourier Transform of a complex sequence.
IFFT_BOX	1484	The inverse Discrete Fourier Transform of several complex or real sequences.
IFNAN(X)	1686	Checks if a value is NaN (not a number).
IICSR	1627	Compares two character strings using the ASCII collating sequence but without regard to case.
IIDEX	1629	Determines the position in a string at which a given character sequence begins without regard to case.
IIMAX	1323	Finds the smallest index of the maximum component of a integer vector.
IIMIN	1323	Finds the smallest index of the minimum of an integer vector.

IMACH	1683	Retrieves integer machine constants.
INLAP	1078	Computes the inverse Laplace transform of a complex function.
ISAMAX	1374	Finds the smallest index of the component of a single-precision vector having maximum absolute value.
ISAMIN	1374	Finds the smallest index of the component of a single-precision vector having minimum absolute value.
ISET	1318	Sets the components of a vector to a scalar, all integer.
ISMAX	1374	Finds the smallest index of the component of a single-precision vector having maximum value.
ISMIN	1374	Finds the smallest index of the component of a single-precision vector having minimum value.
ISNAN	1485	This is a generic logical function used to test scalars or arrays for occurrence of an IEEE 754 Standard format of floating point (ANSI/IEEE 1985) NaN, or not-a-number.
ISRCH	1620	Searches a sorted integer vector for a given integer and return its index.
ISUB	1319	Subtracts each component of a vector from a scalar, $x \leftarrow a - x$, all integer.
ISUM	1322	Sums the values of an integer vector.
ISWAP	1320	Interchanges vectors x and y, both integer.
IUMAG	1658	Sets or retrieves MATH/LIBRARY integer options.
IVMRK	844	Solves an initial-value problem $y' = f(t, y)$ for ordinary differential equations using Runge-Kutta pairs of various orders.
IVPAG	854	Solves an initial-value problem for ordinary differential equations using either Adams-Moulton's or Gear's BDF method.
IVPRK	837	Solves an initial-value problem for ordinary differential equations using the Runge-Kutta-Verner fifth-order and sixth-order method.
IWKCIN	1701	Initializes bookkeeping locations describing the character workspace stack.
IWKIN	1700	Initializes bookkeeping locations describing the workspace stack.
JCGRC	365	Solves a real symmetric definite linear system using the Jacobi preconditioned conjugate gradient method with reverse communication.

LCHRG	406	Computes the Cholesky decomposition of a symmetric positive semidefinite matrix with optional column pivoting.
LCLSQ	388	Solves a linear least-squares problem with linear constraints.
LCONF	1310	Minimizes a general objective function subject to linear equality/inequality constraints.
LCONG	1316	Minimizes a general objective function subject to linear equality/inequality constraints.
LDNCH	412	Downdates the R^TR Cholesky factorization of a real symmetric positive definite matrix after a rank-one matrix is removed.
LFCCB	262	Computes the LU factorization of a complex matrix in band storage mode and estimate its L_1 condition number.
LFCCG	108	Computes the LU factorization of a complex general matrix and estimate its L_1 condition number.
LFCCT	132	Estimates the condition number of a complex triangular matrix.
LFCDH	179	Computes the $R^H R$ factorization of a complex Hermitian positive definite matrix and estimate its L_1 condition number.
LFCDS	143	Computes the $R^T R$ Cholesky factorization of a real symmetric positive definite matrix and estimate its L_1condition number.
LFCHF	197	Computes the $U DU^H$ factorization of a complex Hermitian matrix and estimate its L_1 condition number.
LFCQH	284	Computes the $R^H R$ factorization of a complex Hermitian positive definite matrix in band Hermitian storage mode and estimate its L_1 condition number.
LFCQS	240	Computes the $R^T R$ Cholesky factorization of a real symmetric positive definite matrix in band symmetric storage mode and estimate its L_1 condition number.
LFCRB	219	Computes the LU factorization of a real matrix in band storage mode and estimate its L_1 condition number.
LFCRG	89	Computes the LU factorization of a real general matrix and estimate its L_1 condition number.
LFCRT	125	Estimates the condition number of a real triangular matrix.

LFCSF	162	Computes the $U DU^T$ factorization of a real symmetric matrix and estimate its L_1 condition number.
LFDCB	274	Computes the determinant of a complex matrix given the LU factorization of the matrix in band storage mode.
LFDCG	119	Computes the determinant of a complex general matrix given the LU factorization of the matrix.
LFDCT	134	Computes the determinant of a complex triangular matrix.
LFDDH	190	Computes the determinant of a complex Hermitian positive definite matrix given the $R^H R$ Cholesky factorization of the matrix.
LFDDS	153	Computes the determinant of a real symmetric positive definite matrix given the $R^H R$ Cholesky factorization of the matrix.
LFDHF	207	Computes the determinant of a complex Hermitian matrix given the $U DU^H$ factorization of the matrix.
LFDQH	295	Computes the determinant of a complex Hermitian positive definite matrix given the $R^H R$ Cholesky factorization in band Hermitian storage mode.
LFDQS	250	Computes the determinant of a real symmetric positive definite matrix given the $R^T R$ Cholesky factorization of the band symmetric storage mode.
LFDRB	230	Computes the determinant of a real matrix in band storage mode given the LU factorization of the matrix.
LFDRG	99	Computes the determinant of a real general matrix given the LU factorization of the matrix.
LFDRT	127	Computes the determinant of a real triangular matrix.
LFDSF	172	Computes the determinant of a real symmetric matrix given the $U DU^T$ factorization of the matrix.
LFICB	270	Uses iterative refinement to improve the solution of a complex system of linear equations in band storage mode.
LFICG	116	Uses iterative refinement to improve the solution of a complex general system of linear equations.
LFIDH	187	Uses iterative refinement to improve the solution of a complex Hermitian positive definite system of linear equations.
LFIDS	150	Uses iterative refinement to improve the solution of a real symmetric positive definite system of linear equations.

LFIHF	204	Uses iterative refinement to improve the solution of a complex Hermitian system of linear equations.
LFIQH	292	Uses iterative refinement to improve the solution of a complex Hermitian positive definite system of linear equations in band Hermitian storage mode.
LFIQS	247	Uses iterative refinement to improve the solution of a real symmetric positive definite system of linear equations in band symmetric storage mode.
LFIRB	227	Uses iterative refinement to improve the solution of a real system of linear equations in band storage mode.
LFIRG	96	Uses iterative refinement to improve the solution of a real general system of linear equations.
LFISF	169	Uses iterative refinement to improve the solution of a real symmetric system of linear equations.
LFSCB	268	Solves a complex system of linear equations given the LU factorization of the coefficient matrix in band storage mode.
LFSCG	114	Solves a complex general system of linear equations given the LU factorization of the coefficient matrix.
LFSDH	184	Solves a complex Hermitian positive definite system of linear equations given the $R^H R$ factorization of the coefficient matrix.
LFSDS	148	Solves a real symmetric positive definite system of linear equations given the $R^T R$ Choleksy factorization of the coefficient matrix.
LFSHF	202	Solves a complex Hermitian system of linear equations given the $U DU^H$ factorization of the coefficient matrix.
LFSQH	290	Solves a complex Hermitian positive definite system of linear equations given the factorization of the coefficient matrix in band Hermitian storage mode.
LFSQS	245	Solves a real symmetric positive definite system of linear equations given the factorization of the coefficient matrix in band symmetric storage mode.
LFSRB	225	Solves a real system of linear equations given the LU factorization of the coefficient matrix in band storage mode.
LFSRG	94	Solves a real general system of linear equations given the LU factorization of the coefficient matrix.
LFSSF	167	Solves a real symmetric system of linear equations given the $U DU^T$ factorization of the coefficient matrix.

LFSXD	336	Solves a real sparse symmetric positive definite system of linear equations, given the Cholesky factorization of the coefficient matrix.
LFSXG	306	Solves a sparse system of linear equations given the *LU* factorization of the coefficient matrix.
LFSZD	349	Solves a complex sparse Hermitian positive definite system of linear equations, given the Cholesky factorization of the coefficient matrix.
LFSZG	319	Solves a complex sparse system of linear equations given the *LU* factorization of the coefficient matrix.
LFTCB	265	Computes the *LU* factorization of a complex matrix in band storage mode.
LFTCG	111	Computes the *LU* factorization of a complex general matrix.
LFTDH	182	Computes the $R^H R$ factorization of a complex Hermitian positive definite matrix.
LFTDS	146	Computes the $R^T R$ Cholesky factorization of a real symmetric positive definite matrix.
LFTHF	200	Computes the $U DU^H$ factorization of a complex Hermitian matrix.
LFTQH	288	Computes the $R^H R$ factorization of a complex Hermitian positive definite matrix in band Hermitian storage mode.
LFTQS	243	Computes the $R^T R$ Cholesky factorization of a real symmetric positive definite matrix in band symmetric storage mode.
LFTRB	222	Computes the *LU* factorization of a real matrix in band storage mode.
LFTRG	92	Computes the *LU* factorization of a real general matrix.
LFTSF	164	Computes the $U DU^T$ factorization of a real symmetric matrix.
LFTXG	301	Computes the *LU* factorization of a real general sparse matrix.
LFTZG	314	Computes the *LU* factorization of a complex general sparse matrix.
LINCG	121	Computes the inverse of a complex general matrix.
LINCT	136	Computes the inverse of a complex triangular matrix.
LINDS	154	Computes the inverse of a real symmetric positive definite matrix.

LINRG	101	Computes the inverse of a real general matrix.
LINRT	128	Computes the inverse of a real triangular matrix.
LIN_EIG_GEN	439	Computes the eigenvalues of a self-adjoint matrix, *A*.
LIN_EIG_SELF	432	Computes the eigenvalues of a self-adjoint matrix, *A*.
LIN_GEIG_SELF	448	Computes the generalized eigenvalues of an $n \times n$ matrix pencil, $Av = \lambda Bv$.
LIN_SOL_GEN	9	Solves a general system of linear equations $Ax = b$.
LIN_SOL_LSQ	27	Solves a rectangular system of linear equations $Ax \cong b$, in a least-squares sense.
LIN_SOL_SELF	17	Solves a system of linear equations $Ax = b$, where *A* is a self-adjoint matrix.
LIN_SOL_SVD	36	Solves a rectangular least-squares system of linear equations $Ax \cong b$ using singular value decomposition.
LIN_SOL_TRI	44	Solves multiple systems of linear equations.
LIN_SVD	57	Computes the singular value decomposition (SVD) of a rectangular matrix, *A.*
LNFXD	331	Computes the numerical Cholesky factorization of a sparse symmetrical matrix *A*.
LNFZD	344	Computes the numerical Cholesky factorization of a sparse Hermitian matrix *A*.
LQERR	396	Accumulates the orthogonal matrix *Q* from its factored form given the *QR* factorization of a rectangular matrix *A*.
LQRRR	392	Computes the *QR* decomposition, $AP = QR$, using Householder transformations.
LQRRV	381	Computes the least-squares solution using Householder transformations applied in blocked form.
LQRSL	398	Computes the coordinate transformation, projection, and complete the solution of the least-squares problem $Ax = b$.
LSACB	257	Solves a complex system of linear equations in band storage mode with iterative refinement.
LSACG	103	Solves a complex general system of linear equations with iterative refinement.
LSADH	173	Solves a Hermitian positive definite system of linear equations with iterative refinement.
LSADS	138	Solves a real symmetric positive definite system of linear equations with iterative refinement.

LSAHF 191 Solves a complex Hermitian system of linear equations with iterative refinement.

LSAQH 276 Solves a complex Hermitian positive definite system of linear equations in band Hermitian storage mode with iterative refinement.

LSAQS 232 Solves a real symmetric positive definite system of linear equations in band symmetric storage mode with iterative refinement.

LSARB 213 Solves a real system of linear equations in band storage mode with iterative refinement.

LSARG 83 Solves a real general system of linear equations with iterative refinement.

LSASF 156 Solves a real symmetric system of linear equations with iterative refinement.

LSBRR 385 Solves a linear least-squares problem with iterative refinement.

LSCXD 327 Performs the symbolic Cholesky factorization for a sparse symmetric matrix using a minimum degree ordering or a userspecified ordering, and set up the data structure for the numerical Cholesky factorization.

LSGRR 424 Computes the generalized inverse of a real matrix.

LSLCB 259 Solves a complex system of linear equations in band storage mode without iterative refinement.

LSLCC 356 Solves a complex circulant linear system.

LSLCG 106 Solves a complex general system of linear equations without iterative refinement.

LSLCQ 253 Computes the *LDU* factorization of a complex tridiagonal matrix *A* using a cyclic reduction algorithm.

LSLCR 211 Computes the *LDU* factorization of a real tridiagonal matrix *A* using a cyclic reduction algorithm.

LSLCT 130 Solves a complex triangular system of linear equations.

LSLDH 176 Solves a complex Hermitian positive definite system of linear equations without iterative refinement.

LSLDS 140 Solves a real symmetric positive definite system of linear equations without iterative refinement.

LSLHF 194 Solves a complex Hermitian system of linear equations without iterative refinement.

LSLPB	237	Computes the $R^T DR$ Cholesky factorization of a real symmetric positive definite matrix A in codiagonal band symmetric storage mode. Solve a system $Ax = b$.
LSLQB	281	Computes the $R^H DR$ Cholesky factorization of a complex hermitian positive-definite matrix A in codiagonal band hermitian storage mode. Solve a system $Ax = b$.
LSLQH	279	Solves a complex Hermitian positive definite system of linearequations in band Hermitian storage mode without iterative refinement.
LSLQS	234	Solves a real symmetric positive definite system of linear equations in band symmetric storage mode without iterative refinement.
LSLRB	216	Solves a real system of linear equations in band storage mode without iterative refinement.
LSLRG	85	Solves a real general system of linear equations without iterative refinement.
LSLRT	123	Solves a real triangular system of linear equations.
LSLSF	159	Solves a real symmetric system of linear equations without iterative refinement.
LSLTC	354	Solves a complex Toeplitz linear system.
LSLTO	352	Solves a real Toeplitz linear system.
LSLTQ	252	Solves a complex tridiagonal system of linear equations.
LSLTR	209	Solves a real tridiagonal system of linear equations.
LSLXD	323	Solves a sparse system of symmetric positive definite linear algebraic equations by Gaussian elimination.
LSLXG	297	Solves a sparse system of linear algebraic equations by Gaussian elimination.
LSLZD	340	Solves a complex sparse Hermitian positive definite system of linear equations by Gaussian elimination.
LSLZG	309	Solves a complex sparse system of linear equations by Gaussian elimination.
LSQRR	378	Solves a linear least-squares problem without iterative refinement.
LSVCR	419	Computes the singular value decomposition of a complex matrix.
LSVRR	415	Computes the singular value decomposition of a real matrix.

LUPCH	409	Updates the R^TR Cholesky factorization of a real symmetric positive definite matrix after a rank-one matrix is added.
LUPQR	402	Computes an updated *QR* factorization after the rank-one matrix αxy^T is added.
MCRCR	1423	Multiplies two complex rectangular matrices, *AB*.
MOLCH	946	Solves a system of partial differential equations of the form $u_t = f(x, t, u, u_x, u_{xx})$ using the method of lines. The solution is represented with cubic Hermite polynomials.
MRRRR	1421	Multiplies two real rectangular matrices, *AB*.
MUCBV	1436	Multiplies a complex band matrix in band storage mode by a complex vector.
MUCRV	1435	Multiplies a complex rectangular matrix by a complex vector.
MURBV	1433	Multiplies a real band matrix in band storage mode by a real vector.
MURRV	1431	Multiplies a real rectangular matrix by a vector.
MXTXF	1415	Computes the transpose product of a matrix, A^TA.
MXTYF	1416	Multiplies the transpose of matrix *A* by matrix *B*, A^TB.
MXYTF	1418	Multiplies a matrx *A* by the transpose of a matrix *B*, AB^T.
NAN	1486	Returns, as a scalar function, a value corresponding to the IEEE 754 Standard format of floating point (ANSI/IEEE 1985) for NaN. .
N1RTY	1680	Retrieves an error type for the most recently called IMSL routine.
NDAYS	1634	Computes the number of days from January 1, 1900, to the given date.
NDYIN	1636	Gives the date corresponding to the number of days since January 1, 1900.
NEQBF	1169	Solves a system of nonlinear equations using factored secant update with a finite-difference approximation to the Jacobian.
NEQBJ	1174	Solves a system of nonlinear equations using factored secant update with a user-supplied Jacobian.
NEQNF	1162	Solves a system of nonlinear equations using a modified Powell hybrid algorithm and a finite-difference approximation to the Jacobian.

NEQNJ	1165	Solves a system of nonlinear equations using a modified Powell hybrid algorithm with a user-supplied Jacobian.
NNLPF	1323	Uses a sequential equality constrained QP method.
NNLPG	1329	Uses a sequential equality constrained QP method.
NORM	1487	Computes the norm of a rank-1 or rank-2 array. For rank-3 arrays, the norms of each rank-2 array, in dimension 3, are computed.
NR1CB	1449	Computes the 1-norm of a complex band matrix in band storage mode.
NR1RB	1447	Computes the 1-norm of a real band matrix in band storage mode.
NR1RR	1444	Computes the 1-norm of a real matrix.
NR2RR	1446	Computes the Frobenius norm of a real rectangular matrix.
NRIRR	1443	Computes the infinity norm of a real matrix.
OPERATOR: .h.	1472	Computes transpose and conjugate transpose of a matrix.
OPERATOR: .hx.	1471	Computes matrix-vector and matrix-matrix products.
OPERATOR:.i.	1473	Computes the inverse matrix, for square non-singular matrices.
OPERATOR:.ix.	1474	Computes the inverse matrix times a vector or matrix for square non-singular matrices.
OPERATOR:..t.	1472	Computes transpose and conjugate transpose of a matrix.
OPERATOR:.tx.	1471	Computes matrix-vector and matrix-matrix products.
OPERATOR:.x.	1471	Computes matrix-vector and matrix-matrix products..
OPERATOR:..xh.	1471	Computes matrix-vector and matrix-matrix products.
OPERATOR:..xi.	1474	Computes the inverse matrix times a vector or matrix for square non-singular matrices.
OPERATORS:.xt.	1471	Computes matrix-vector and matrix-matrix products.
ORTH	1488	Orthogonalizes the columns of a rank-2 or rank-3 array.
PCGRC	359	Solves a real symmetric definite linear system using a preconditioned conjugate gradient method with reverse communication.
PARALLEL_NONNEGATIVE_LSQ	67	Solves a linear, non-negative constrained least-squares system.
PARALLEL_BOUNDED_LSQ	75	Solves a linear least-squares system with bounds on the unknowns.
PDE_1D_MG	913	Method of lines with Variable Griddings.

PERMA	1602	Permutes the rows or columns of a matrix.
PERMU	1600	Rearranges the elements of an array as specified by a permutation.
PGOPT	1599	Sets or retrieves page width and length for printing.
PLOTP	1664	Prints a plot of up to 10 sets of points.
POLRG	1429	Evaluates a real general matrix polynomial.
PP1GD	687	Evaluates the derivative of a piecewise polynomial on a grid.
PPDER	684	Evaluates the derivative of a piecewise polynomial.
PPITG	690	Evaluates the integral of a piecewise polynomial.
PPVAL	681	Evaluates a piecewise polynomial.
PRIME	1668	Decomposes an integer into its prime factors.
QAND	806	Integrates a function on a hyper-rectangle.
QCOSB	1041	Computes a sequence from its cosine Fourier coefficients with only odd wave numbers.
QCOSF	1039	Computes the coefficients of the cosine Fourier transform with only odd wave numbers.
QCOSI	1043	Computes parameters needed by QCOSF and QCOSB.
QD2DR	699	Evaluates the derivative of a function defined on a rectangular grid using quadratic interpolation.
QD2VL	696	Evaluates a function defined on a rectangular grid using quadratic interpolation.
QD3DR	705	Evaluates the derivative of a function defined on a rectangular three-dimensional grid using quadratic interpolation.
QD3VL	702	Evaluates a function defined on a rectangular three-dimensional grid using quadratic interpolation.
QDAG	775	Integrates a function using a globally adaptive scheme based on Gauss-Kronrod rules.
QDAGI	782	Integrates a function over an infinite or semi-infinite interval.
QDAGP	779	Integrates a function with singularity points given.
QDAGS	772	Integrates a function (which may have endpoint singularities).
QDAWC	796	Integrates a function F(X)/(X – C) in the Cauchy principal value sense.
QDAWF	789	Computes a Fourier integral.

QDAWO	785	Integrates a function containing a sine or a cosine.
QDAWS	793	Integrates a function with algebraic-logarithmic singularities.
QDDER	694	Evaluates the derivative of a function defined on a set of points using quadratic interpolation.
QDNG	799	Integrates a smooth function using a nonadaptive rule.
QDVAL	692	Evaluates a function defined on a set of points using quadratic interpolation.
QMC	809	Integrates a function over a hyperrectangle using a quasi-Monte Carlo method.
QPROG	1307	Solves a quadratic programming problem subject to linear equality/inequality constraints.
QSINB	1034	Computes a sequence from its sine Fourier coefficients with only odd wave numbers.
QSINF	1032	Computes the coefficients of the sine Fourier transform with only odd wave numbers.
QSINI	1037	Computes parameters needed by QSINF and QSINB.
RAND	1489	Computes a scalar, rank-1, rank-2 or rank-3 array of random numbers.
RAND_GEN	1639	Generates a rank-1 array of random numbers.
RANK	1490	Computes the mathematical rank of a rank-2 or rank-3 array.
RATCH	764	Computes a rational weighted Chebyshev approximation to a continuous function on an interval.
RCONV	1059	Computes the convolution of two real vectors.
RCORL	1068	Computes the correlation of two real vectors.
RCURV	716	Fits a polynomial curve using least squares.
RECCF	818	Computes recurrence coefficients for various monic polynomials.
RECQR	821	Computes recurrence coefficients for monic polynomials given a quadrature rule.
RLINE	713	Fits a line to a set of data points using least squares.
RNGET	1648	Retrieves the current value of the seed used in the IMSL random number generators.
RNOPT	1650	Selects the uniform (0, 1) multiplicative congruential pseudorandom number generator.
RNSET	1649	Initializes a random seed for use in the IMSL random number generators.

RNUN	1653	Generates pseudorandom numbers from a uniform (0, 1) distribution.
RNUNF	1651	Generates a pseudorandom number from a uniform (0, 1) distribution.
SADD	1370	Adds a scalar to each component of a vector, $x \leftarrow x + a$, all single precision.
SASUM	1373	Sums the absolute values of the components of a single-precision vector.
SAXPY	1370	Computes the scalar times a vector plus a vector, $y \leftarrow ax + y$, all single precision.
ScaLaPACK_READ	1545	Reads matrix data from a file and transmits it into the two-dimensional block-cyclic form required by *ScaLAPACK* routines.
ScaLaPACK_WRITE	1547	Writes the matrix data to a file.
SCASUM	1322	Sums the absolute values of the real part together with the absolute values of the imaginary part of the components of a complex vector.
SCNRM2	1322	Computes the Euclidean norm of a complex vector.
SCOPY	1369	Copies a vector *x* to a vector *y*, both single precision.
SDDOTA	1321	Computes the sum of a single-precision scalar, a single-precision dot product and the double-precision accumulator, which is set to the result $\mathrm{ACC} \leftarrow \mathrm{ACC} + a + x^T y$.
SDDOTI	1372	Computes the sum of a single-precision scalar plus a singleprecision dot product using a double-precision accumulator, which is set to the result $\mathrm{ACC} \leftarrow a + x^T y$.
SDOT	1370	Computes the single-precision dot product $x^T y$.
SDSDOT	1371	Computes the sum of a single-precision scalar and a single precision dot product, $a + x^T y$, using a double-precision accumulator.
SGBMV	1381	Computes one of the matrix-vector operations: $y \leftarrow \alpha Ax + \beta y,$ or $y \leftarrow \alpha A^T x + \beta y$, where *A* is a matrix stored in band storage mode.
SGEMM	1385	Computes one of the matrix-matrix operations: $C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha A^T B + \beta C, C \leftarrow \alpha AB^T$. $+\beta C$, or $C \leftarrow \alpha A^T B^T + \beta C$

SGEMV	1381	Computes one of the matrix-vector operations: $y \leftarrow \alpha Ax + \beta y$, or $y \leftarrow \alpha A^T x + \beta y$,
SGER	1383	Computes the rank-one update of a real general matrix: $A \leftarrow A + \alpha xy^T$.
SHOW	1571	Prints rank-1 or rank-2 arrays of numbers in a readable format.
SHPROD	1372	Computes the Hadamard product of two single-precision vectors.
SINLP	1081	Computes the inverse Laplace transform of a complex function.
SLCNT	986	Calculates the indices of eigenvalues of a Sturm-Liouville problem with boundary conditions (at regular points) in a specified subinterval of the real line, [α, β].
SLEIG	973	Determines eigenvalues, eigenfunctions and/or spectral density functions for Sturm-Liouville problems in the form with boundary conditions (at regular points).
SLPRS	1301	Solves a sparse linear programming problem via the revised simplex algorithm.
SNRM2	1373	Computes the Euclidean length or L_2 norm of a single-precision vector.
SORT_REAL	1604	Sorts a rank-1 array of real numbers x so the y results are algebraically nondecreasing, $y_1 \le y_2 \le \dots y_n$.
SPLEZ	618	Computes the values of a spline that either interpolates or fits user-supplied data.
SPLINE_CONSTRAINTS	562	Returns the derived type array result.
SPLINE_FITTING	564	Weighted least-squares fitting by B-splines to discrete One-Dimensional data is performed.
SPLINE_VALUES	563	Returns an array result, given an array of input
SPRDCT	1373	Multiplies the components of a single-precision vector.
SRCH	1618	Searches a sorted vector for a given scalar and return its index.
SROT	1375	Applies a Givens plane rotation in single precision.
SROTG	1374	Constructs a Givens plane rotation in single precision.
SROTM	1377	Applies a modified Givens plane rotation in single precision.
SROTMG	1376	Constructs a modified Givens plane rotation in single precision.

SSBMV	1382	Computes the matrix-vector operation $y \leftarrow \alpha Ax + \beta y$, where A is a symmetric matrix in band symmetric storage mode.
SSCAL	1369	Multiplies a vector by a scalar, $y \leftarrow ay$, both single precision.
SSET	1369	Sets the components of a vector to a scalar, all single precision.
SSRCH	1622	Searches a character vector, sorted in ascending ASCII order, for a given string and return its index.
SSUB	1370	Subtracts each component of a vector from a scalar, $x \leftarrow a - x$, all single precision.
SSUM	1372	Sums the values of a single-precision vector.
SSWAP	1370	Interchanges vectors x and y, both single precision.
SSYMM	1385	Computes one of the matrix-matrix operations: $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$, where A is a symmetric matrix and B and C are m by n matrices.
SSYMV	1382	Computes the matrix-vector operation $y \leftarrow \alpha Ax + \beta y$, where A is a symmetric matrix.
SSYR	1384	Computes the rank-one update of a real symmetric matrix: $A \leftarrow A + \alpha xx^T$.
SSYR2	1384	Computes the rank-two update of a real symmetric matrix: $A \leftarrow A + \alpha xy^T + \alpha yx^T$.
SSYR2K	1386	Computes one of the symmetric rank $2k$ operations: $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$ or $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$, where C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.
SSYRK	1386	Computes one of the symmetric rank k operations: $C \leftarrow \alpha AA^T + \beta C$ or $C \leftarrow \alpha A^T A + \beta C$, where C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.
STBMV	1382	Computes one of the matrix-vector operations: $x \leftarrow Ax$ or $x \leftarrow A^T x$, where A is a triangular matrix in band storage mode.

STBSV	1383	Solves one of the triangular systems: $x \leftarrow A^{-1}x$ or $x \leftarrow \left(A^{-1}\right)^T x$, where A is a triangular matrix in band storage mode.
STRMM	1387	Computes one of the matrix-matrix operations: $B \leftarrow \alpha AB$, $B \leftarrow \alpha A^T B$ or $B \leftarrow \alpha BA$, $B \leftarrow \alpha BA^T$, where B is an m by n matrix and A is a triangular matrix.
STRMV	1382	Computes one of the matrix-vector operations: $x \leftarrow Ax$ or $x \leftarrow A^T x$, where A is a triangular matrix.
STRSM	1387	Solves one of the matrix equations: $B \leftarrow \alpha A^{-1}B$, $B \leftarrow \alpha BA^{-1}$, $B \leftarrow \alpha\left(A^{-1}\right)^T B$, or $B \leftarrow \alpha B\left(A^{-1}\right)^T$ where B is an m by n matrix and A is a triangular matrix.
STRSV	1383	Solves one of the triangular linear systems: $x \leftarrow A^{-1}x$ or $x \leftarrow \left(A^{-1}\right)^T x$ where A is a triangular matrix.
SUMAG	1664	Sets or retrieves MATH/LIBRARY single-precision options.
SURF	710	Computes a smooth bivariate interpolant to scattered data that is locally a quintic polynomial in two variables.
SURFACE_CONSTRAINTS	574	Returns the derived type array result given optional input.
SURFACE_FITTING	577	Weighted least-squares fitting by tensor product B-splines to discrete two-dimensional data is performed.
SURFACE_VALUES	575	Returns a tensor product array result, given two arrays of independent variable values.
SVCAL	1369	Multiplies a vector by a scalar and store the result in another vector, $y \leftarrow ax$, all single precision.
SVD	1491	Computes the singular value decomposition of a rank-2 or rank-3 array, $A = USV^T$.
SVIBN	1615	Sorts an integer array by nondecreasing absolute value.
SVIBP	1617	Sorts an integer array by nondecreasing absolute value and returns the permutation that rearranges the array.
SVIGN	1610	Sorts an integer array by algebraically increasing value.

SVIGP	1611	Sorts an integer array by algebraically increasing value and returns the permutation that rearranges the array.
SVRBN	1612	Sorts a real array by nondecreasing absolute value.
SVRBP	1614	Sorts a real array by nondecreasing absolute value and returns the permutation that rearranges the array.
SVRGN	1607	Sorts a real array by algebraically increasing value.
SVRGP	1608	Sorts a real array by algebraically increasing value and returns the permutation that rearranges the array.
SXYZ	1372	Computes a single-precision *xyz* product.
TDATE	1633	Gets today's date.
TIMDY	1632	Gets time of day.
TRNRR	1413	Transposes a rectangular matrix.
TWODQ	801	Computes a two-dimensional iterated integral.
UMACH	1688	Sets or retrieves input or output device unit numbers.
UMAG	1661	Handles MATH/LIBRARY and STAT/LIBRARY type `REAL` and double precision options.
UMCGF	1219	Minimizes a function of `N` variables using a conjugate gradient algorithm and a finite-difference gradient.
UMCGG	1223	Minimizes a function of `N` variables using a conjugate gradient algorithm and a user-supplied gradient.
UMIAH	1213	Minimizes a function of `N` variables using a modified Newton method and a user-supplied Hessian.
UMIDH	1208	Minimizes a function of `N` variables using a modified Newton method and a finite-difference Hessian.
UMINF	1196	Minimizes a function of `N` variables using a quasi-New method and a finite-difference gradient.
UMING	1202	Minimizes a function of `N` variables using a quasi-New method and a user-supplied gradient.
UMPOL	1227	Minimizes a function of `N` variables using a direct search polytope algorithm.
UNIT	1492	Normalizes the columns of a rank-2 or rank-3 array so each has Euclidean length of value one.
UNLSF	1231	Solves a nonlinear least squares problem using a modified Levenberg-Marquardt algorithm and a finite-difference Jacobian.
UNLSJ	1237	Solves a nonlinear least squares problem using a modified Levenberg-Marquardt algorithm and a user-supplied Jacobian.

UVMGS	1193	Finds the minimum point of a nonsmooth function of a single variable.
UVMID	1189	Finds the minimum point of a smooth function of a single variable using both function evaluations and first derivative evaluations.
UVMIF	1186	Finds the minimum point of a smooth function of a single variable using only function evaluations.
VCONC	1457	Computes the convolution of two complex vectors.
VCONR	1455	Computes the convolution of two real vectors.
VERML	1638	Obtains IMSL MATH/LIBRARY-related version, system and license numbers.
WRCRL	1588	Prints a complex rectangular matrix with a given format and labels.
WRCRN	1586	Prints a complex rectangular matrix with integer row and column labels.
WRIRL	1583	Prints an integer rectangular matrix with a given format and labels.
WRIRN	1581	Prints an integer rectangular matrix with integer row and column labels.
WROPT	1591	Sets or retrieves an option for printing a matrix.
WRRRL	1577	Prints a real rectangular matrix with a given format and labels.
WRRRN	1575	Prints a real rectangular matrix with integer row and column labels.
ZANLY	1153	Finds the zeros of a univariate complex function using Müller’s method.
ZBREN	1156	Finds a zero of a real function that changes sign in a given interval.
ZPLRC	1148	Finds the zeros of a polynomial with real coefficients using Laguerre’s method.
ZPOCC	1152	Finds the zeros of a polynomial with complex coefficients using the Jenkins-Traub three-stage algorithm.
ZPORC	1150	Finds the zeros of a polynomial with real coefficients using the Jenkins-Traub three-stage algorithm.
ZQADD	1460	Adds a double complex scalar to the accumulator in extended precision.
ZQINI	1460	Initializes an extended-precision complex accumulator to a double complex scalar.

ZQMUL	1460	Multiplies double complex scalars using extended precision.
ZQSTO	1460	Stores a double complex approximation to an extended-precision complex scalar.
ZREAL	1159	Finds the real zeros of a real function using Müller’s method.

Appendix C: References

Aird and Howell

Aird, Thomas J., and Byron W. Howell (1991), IMSL Technical Report 9103, IMSL, Houston.

Aird and Rice

Aird, T.J., and J.R. Rice (1977), Systematic search in high dimensional sets, *SIAM Journal on Numerical Analysis*, **14**, 296–312.

Akima

Akima, H. (1970), A new method of interpolation and smooth curve fitting based on local procedures, *Journal of the ACM*, **17**, 589−602.

Akima, H. (1978), A method of bivariate interpolation and smooth surface fitting for irregularly distributed data points, *ACM Transactions on Mathematical Software*, **4**, 148−159.

Arushanian et al.

Arushanian, O.B., M.K. Samarin, V.V. Voevodin, E.E. Tyrtyshikov, B.S. Garbow, J.M. Boyle, W.R. Cowell, and K.W. Dritz (1983), *The TOEPLITZ Package Users' Guide*, Argonne National Laboratory, Argonne, Illinois.

Ashcraft

Ashcraft, C. (1987), *A vector implementation of the multifrontal method for large sparse, symmetric positive definite linear systems*, Technical Report ETA-TR-51, Engineering Technology Applications Division, Boeing Computer Services, Seattle, Washington.

Ashcraft et al.

Ashcraft, C., R.Grimes, J. Lewis, B. Peyton, and H. Simon (1987), Progress in sparse matrix methods for large linear systems on vector supercomputers. *Intern. J. Supercomputer Applic.*, **1**(**4**), 10–29.

Atkinson

Atkinson, Ken (1978), *An Introduction to Numerical Analysis*, John Wiley & Sons, New York.

Atchison and Hanson

Atchison, M.A., and R.J. Hanson (1991), *An Options Manager for the IMSL Fortran 77 Libraries*, Technical Report 9101, IMSL, Houston.

Bischof et al.

Bischof, C., J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, D. Sorensen (1988), LAPACK Working Note #5: Provisional Contents, Argonne National Laboratory Report ANL-88-38, Mathematics and Computer Science.

Bjorck

Bjorck, Ake (1967), Iterative refinement of linear least squares solutions I, BIT, **7**, 322–337.

Bjorck, Ake (1968), Iterative refinement of linear least squares solutions II, BIT, **8**, 8–30.

Boisvert (1984)

Boisvert, Ronald (1984), A fourth order accurate fast direct method for the Helmholtz equation, *Elliptic Problem Solvers II*, (edited by G. Birkhoff and A. Schoenstadt), Academic Press, Orlando, Florida, 35–44.

Boisvert, Howe, and Kahaner

Boisvert, Ronald F., Sally E. Howe, and David K. Kahaner (1985), GAMS: A framework for the management of scientific software, *ACM Transactions on Mathematical Software*, **11**, 313–355.

Boisvert, Howe, Kahaner, and Springmann

Boisvert, Ronald F., Sally E. Howe, David K. Kahaner, and Jeanne L. Springmann (1990), *Guide to Available Mathematical Software*, NISTIR 90-4237, National Institute of Standards and Technology, Gaithersburg, Maryland.

Brankin et al.

Brankin, R.W., I. Gladwell, and L.F. Shampine, RKSUITE: a Suite of Runge-Kutta Codes for the Initial Value Problem for ODEs, Softreport 91-1, Mathematics Department, Southern Methodist University, Dallas, Texas, 1991.

Brenan, Campbell, and Petzold

Brenan, K.E., S.L. Campbell, L.R. Petzold (1989), *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, Elseview Science Publ. Co.

Brenner

Brenner, N. (1973), Algorithm 467: Matrix transposition in place [F1], *Communication of ACM*, **16**, 692–694.

Brent

Brent, R.P. (1971), An algorithm with guaranteed convergence for finding a zero of a function, *The Computer Journa*l, **14**, 422–425.

Brent, Richard P. (1973), *Algorithms for Minimization without Derivatives*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Brigham

Brigham, E. Oran (1974), *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, New Jersey.

Cheney

Cheney, E.W. (1966), *Introduction to Approximation Theory*, McGraw-Hill, New York.

Cline et al.

Cline, A.K., C.B. Moler, G.W. Stewart, and J.H. Wilkinson (1979), An estimate for the condition number of a matrix, *SIAM Journal of Numerical Analysis*, **16**, 368−375.

Cody, Fraser, and Hart

Cody, W.J., W. Fraser, and J.F. Hart (1968), Rational Chebyshev approximation using linear equations, *Numerische Mathematik*, **12**, 242–251.

Cohen and Taylor

Cohen, E. Richard, and Barry N. Taylor (1986), *The 1986 Adjustment of the Fundamental Physical Constants*, Codata Bulletin, Pergamon Press, New York.

Cooley and Tukey

Cooley, J.W., and J.W. Tukey (1965), An algorithm for the machine computation of complex Fourier series, *Mathematics of Computation*, **19**, 297–301.

Courant and Hilbert

Courant, R., and D. Hilbert (1962), *Methods of Mathematical Physics, Volume II*, John Wiley & Sons, New York, NY.

Craven and Wahba

Craven, Peter, and Grace Wahba (1979), Smoothing noisy data with spline functions, *Numerische Mathematik*, **31**, 377–403.

Crowe et al.

Crowe, Keith, Yuan-An Fan, Jing Li, Dale Neaderhouser, and Phil Smith (1990), *A direct sparse linear equation solver using linked list storage*, IMSL Technical Report 9006, IMSL, Houston.

Crump

Crump, Kenny S. (1976), Numerical inversion of Laplace transforms using a Fourier series approximation, *Journal of the Association for Computing Machinery*, **23**, 89–96.

Davis and Rabinowitz

Davis, Philip F., and Philip Rabinowitz (1984), *Methods of Numerical Integration*, Academic Press, Orlando, Florida.

de Boor

de Boor, Carl (1978), *A Practical Guide to Splines*, Springer-Verlag, New York.

de Hoog, Knight, and Stokes

de Hoog, F.R., J.H. Knight, and A.N. Stokes (1982), An improved method for numerical inversion of Laplace transforms. *SIAM Journal on Scientific and Statistical Computing*, **3**, 357–366.

Dennis and Schnabel

Dennis, J.E., Jr., and Robert B. Schnabel (1983), *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, Englewood Cliffs, New Jersey.

Dongarra et al.

Dongarra, J.J., and C.B. Moler, (1977) *EISPACK – A package for solving matrix eigenvalue problems*, Argonne National Laboratory, Argonne, Illinois.

Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart (1979), *LINPACK Users' Guide*, SIAM, Philadelphia.

Dongarra, J.J., J. DuCroz, S. Hammarling, R. J. Hanson (1988), An Extended Set of Fortran basic linear algebra subprograms, *ACM Transactions on Mathematical Software*, **14** , 1–17.

Dongarra, J.J., J. DuCroz, S. Hammarling, I. Duff (1990), A set of level 3 basic linear algebra subprograms, *ACM Transactions on Mathematical Software*, **16** , 1–17.

Draper and Smith

Draper, N.R., and H. Smith (1981), *Applied Regression Analysis*, second edition, John Wiley & Sons, New York.

Du Croz et al.

Du Croz, Jeremy, P. Mayes, G. and Radicati (1990), Factorization of band matrices using Level-3 BLAS, *Proceedings of CONPAR 90 VAPP IV, Lecture Notes in Computer Science*, Springer, Berlin, 222.

Duff and Reid

Duff, I.S., and J.K. Reid (1983), The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, **9**, 302–325.

Duff, I.S., and J.K. Reid (1984), The multifrontal solution of unsymmetric sets of linear equations. *SIAM Journal on Scientific and Statistical Computing*, **5**, 633−641.

Duff et al.

Duff, I.S., A.M. Erisman, and J.K. Reid (1986), *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford.

Enright and Pryce

Enright, W.H., and J.D. Pryce (1987), Two FORTRAN packages for assessing initial value methods, *ACM Transactions on Mathematical Software*, **13**, 1−22.

Forsythe

Forsythe, G.E. (1957), Generation and use of orthogonal polynomials for fitting data with a digital computer, *SIAM Journal on Applied Mathematics*, **5**, 74−88.

Fox, Hall, and Schryer

Fox, P.A., A.D. Hall, and N.L. Schryer (1978), The PORT mathematical subroutine library, *ACM Transactions on Mathematical Software*, **4**, 104−126.

Garbow

Garbow, B.S. (1978) `CALGO` Algorithm 535: The QZ algorithm to solve the generalized eigenvalue problem for complex matrices, *ACM Transactions on Mathematical Software*, **4**, 404−410.

Garbow et al.

Garbow, B.S., J.M. Boyle, J.J. Dongarra, and C.B. Moler (1972), *Matrix eigensystem Routines: EISPACK Guide Extension*, Springer-Verlag, New York.

Garbow, B.S., J.M. Boyle, J.J. Dongarra, and C.B. Moler (1977), *Matrix Eigensystem Routines− EISPACK Guide Extension*, Springer-Verlag, New York.

Garbow, B.S., G. Giunta, J.N. Lyness, and A. Murli (1988), Software for an implementation of Weeks’ method for the inverse Laplace transform problem, *ACM Transactions of Mathematical Software*, **14**, 163−170.

Gautschi

Gautschi, Walter (1968), Construction of Gauss-Christoffel quadrature formulas, *Mathematics of Computation*, **22**, 251−270.

Gautschi and Milovanofic

Gautschi, Walter, and Gradimir V. Milovanofic (1985), Gaussian quadrature involving Einstein and Fermi functions with an application to summation of series, *Mathematics of Computation*, **44**, 177−190.

Gay

Gay, David M. (1981), Computing optimal locally constrained steps, *SIAM Journal on Scientific and Statistical Computing*, **2**, 186–197.

Gay, David M. (1983), Algorithm 611: Subroutine for unconstrained minimization using a model/trust-region approach, *ACM Transactions on Mathematical Software*, **9**, 503– 524.

Gear

Gear, C.W. (1971), *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, New Jersey.

Gear and Petzold

Gear, C.W., and Linda R. Petzold (1984), ODE methods for the solutions of differential/algebraic equations, *SIAM Journal Numerical Analysis*, **21**, #4, 716.

George and Liu

George, A., and J.W.H. Liu (1981), *Computer Solution of Large Sparse Positive-definite Systems*, Prentice-Hall, Englewood Cliffs, New Jersey.

Gill et al.

Gill, Philip E., and Walter Murray (1976), *Minimization subject to bounds on the variables*, NPL Report NAC 72, National Physical Laboratory, England.

Gill, Philip E., Walter Murray, and Margaret Wright (1981), *Practical Optimization*, Academic Press, New York.

Gill, P.E., W. Murray, M.A. Saunders, and M.H. Wright (1985), Model building and practical aspects of nonlinear programming, in *Computational Mathematical Programming*, (edited by K. Schittkowski), NATO ASI Series, **15**, Springer-Verlag, Berlin, Germany.

Goldfarb and Idnani

Goldfarb, D., and A. Idnani (1983), A numerically stable dual method for solving strictly convex quadratic programs, *Mathematical Programming*, **27**, 1–33.

Golub

Golub, G.H. (1973), Some modified matrix eigenvalue problems, *SIAM Review*, **15**, 318–334.

Golub and Van Loan

Golub, Gene H., and Charles F. Van Loan (1983), *Matrix Computations*, Johns Hopkins University Press, Baltimore, Maryland.

Golub, Gene H., and Charles F. Van Loan (1989), *Matrix Computations*, 2d ed., Johns Hopkins University Press, Baltimore, Maryland.

Golub and Welsch

Golub, G.H., and J.H. Welsch (1969), Calculation of Gaussian quadrature rules, *Mathematics of Computation*, **23**, 221–230.

Gregory and Karney

Gregory, Robert, and David Karney (1969), *A Collection of Matrices for Testing Computational Algorithms*, Wiley-Interscience, John Wiley & Sons, New York.

Griffin and Redish

Griffin, R., and K.A. Redish (1970), Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **13**, 54.

Grosse

Grosse, Eric (1980), Tensor spline approximation, *Linear Algebra and its Applications*, **34**, 29–41.

Guerra and Tapia

Guerra, V., and R. A. Tapia (1974), *A local procedure for error detection and data smoothing*, MRC Technical Summary Report 1452, Mathematics Research Center, University of Wisconsin, Madison.

Hageman and Young

Hageman, Louis A., and David M.Young (1981), *Applied Iterative Methods*, Academic Press, New York.

Hanson

Hanson, Richard J. (1986), Least squares with bounds and linear constraints, *SIAM Journal Sci. Stat. Computing*, **7**, #3.

Hanson, Richard.J. (1990), *A cyclic reduction solver for the IMSL Mathematics Library*, IMSL Technical Report 9002, IMSL, Houston.

Hanson et al.

Hanson, Richard J., R. Lehoucq, J. Stolle, and A. Belmonte (1990), *Improved performance of certain matrix eigenvalue computations for the IMSL/MATH Library*, IMSL Technical Report 9007, IMSL, Houston.

Hartman

Hartman, Philip (1964) *Ordinary Differential Equations*, John Wiley and Sons, New York, NY.

Hausman

Hausman, Jr., R.F. (1971), *Function Optimization on a Line Segment by Golden Section*, Lawrence Radiation Laboratory, University of California, Livermore.

Hindmarsh

Hindmarsh, A.C. (1974), *GEAR: Ordinary differential equation system solver*, Lawrence Livermore Laboratory Report UCID–30001, Revision 3.

Hull et al.

Hull, T.E., W.H. Enright, and K.R. Jackson (1976), *User's guide for DVERK – A subroutine for solving non-stiff ODEs*, Department of Computer Science Technical Report 100, University of Toronto.

IEEE

ANSI/IEEE Std 754-1985 (1985), *IEEE Standard for Binary Floating-Point Arithmetic*, The IEEE, Inc., New York.

IMSL (1991)

IMSL (1991), IMSL STAT/LIBRARY *User's Manual, Version 2.0*, IMSL, Houston.

Irvine et al.

Irvine, Larry D., Samuel P. Marin, and Philip W. Smith (1986), Constrained interpolation and smoothing, *Constructive Approximation*, **2**, 129–151.

Jenkins

Jenkins, M.A. (1975), Algorithm 493: Zeros of a real polynomial, *ACM Transactions on Mathematical Software*, **1**, 178–189.

Jenkins and Traub

Jenkins, M.A., and J.F. Traub (1970), A three-stage algorithm for real polynomials using quadratic iteration, *SIAM Journal on Numerical Analysis*, **7**, 545–566.

Jenkins, M.A., and J.F. Traub (1970), A three-stage variable-shift iteration for polynomial zeros and its relation to generalized Rayleigh iteration, *Numerische Mathematik*, **14**, 252–263.

Jenkins, M.A., and J.F. Traub (1972), Zeros of a complex polynomial, *Communications of the ACM*, **15**, 97–99.

Kennedy and Gentle

Kennedy, William J., Jr., and James E. Gentle (1980), *Statistical Computing*, Marcel Dekker, New York.

Kershaw

Kershaw, D. (1982), Solution of tridiagonal linear systems and vectorization of the ICCG algorithm on the Cray-1, *Parallel Computations*, Academic Press, Inc., 85-99.

Knuth

Knuth, Donald E. (1973), *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Addison-Wesley Publishing Company, Reading, Mass.

Lawson et al.

Lawson, C.L., R.J. Hanson, D.R. Kincaid, and F.T. Krogh (1979), Basic linear algebra subprograms for Fortran usage, *ACM Transactions on Mathematical Software*, **5**, 308− 323.

Leavenworth

Leavenworth, B. (1960), Algorithm 25: Real zeros of an arbitrary function, *Communications of the ACM*, **3**, 602.

Levenberg

Levenberg, K. (1944), A method for the solution of certain problems in least squares, *Quarterly of Applied Mathematics*, **2**, 164−168.

Lewis et al.

Lewis, P.A. W., A.S. Goodman, and J.M. Miller (1969), A pseudo-random number generator for the System/360, *IBM Systems Journa*l, **8**, 136−146.

Liepman

Liepman, David S. (1964), Mathematical constants, in *Handbook of Mathematical Functions*, Dover Publications, New York.

Liu

Liu, J.W.H. (1986), On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, **12**, 249–264.

Liu, J.W.H. (1987), *A collection of routines for an implementation of the multifrontal method*, Technical Report CS-87-10, Department of Computer Science, York University, North York, Ontario, Canada.

Liu, J.W.H. (1989), The multifrontal method and paging in sparse Cholesky factorization. *ACM Transactions on Mathematical Software*, **15**, 310–325.

Liu, J.W.H. (1990), The multifrontal method for sparse matrix solution: theory and practice, Technical Report CS-90-04, Department of Computer Science, York University, North York, Ontario, Canada.

Liu and Ashcraft

Liu, J., and C. Ashcraft (1987), *A vector implementation of the multifrontal method for large sparse, symmetric positive definite linear systems*, Technical Report ETA-TR-51, Engineering Technology Applications Division, Boeing Computer Services, Seattle, Washington.

Lyness and Giunta

Lyness, J.N. and G. Giunta (1986), A modification of the Weeks Method for numerical inversion of the Laplace transform, *Mathmetics of Computation*, **47**, 313–322.

Madsen and Sincovec

Madsen, N.K., and R.F. Sincovec (1979), Algorithm 540: PDECOL, General collocation software for partial differential equations, *ACM Transactions on Mathematical Software*, **5**, #3, 326-351.

Marquardt

Marquardt, D. (1963), An algorithm for least-squares estimation of nonlinear parameters, *SIAM Journal on Applied Mathematics*, **11**, 431–441.

Martin and Wilkinson

Martin, R.S., and J.W. Wilkinson (1968), Reduction of the symmetric eigenproblem $Ax = \lambda Bx$ and related problems to standard form, *Numerische Mathematik*, **11**, 99–119.

Micchelli et al.

Micchelli, C.A., T.J. Rivlin, and S. Winograd (1976), The optimal recovery of smooth functions, *Numerische Mathematik*, **26**, 279–285

Micchelli, C.A., Philip W. Smith, John Swetits, and Joseph D. Ward (1985), Constrained L_p approximation, *Constructive Approximation*, **1**, 93–102.

Moler and Stewart

Moler, C., and G.W. Stewart (1973), An algorithm for generalized matrix eigenvalue problems, *SIAM Journal on Numerical Analysis*, **10**, 241–256.

More et al.

More, Jorge, Burton Garbow, and Kenneth Hillstrom (1980), *User guide for MINPACK-1*, Argonne National Labs Report ANL-80-74, Argonne, Illinois.

Muller

Muller, D.E. (1956), A method for solving algebraic equations using an automatic computer, *Mathematical Tables and Aids to Computation*, **10**, 208–215.

Murtagh

Murtagh, Bruce A. (1981), *Advanced Linear Programming: Computation and Practice*, McGraw-Hill, New York.

Murty

Murty, Katta G. (1983), *Linear Programming*, John Wiley and Sons, New York.

Nelder and Mead

Nelder, J.A., and R. Mead (1965), A simplex method for function minimization, *Computer Journa*l **7**, 308–313.

Neter and Wasserman

Neter, John, and William Wasserman (1974), *Applied Linear Statistical Models*, Richard D. Irwin, Homewood, Ill.

Park and Miller

Park, Stephen K., and Keith W. Miller (1988), Random number generators: good ones are hard to find, *Communications of the ACM*, **31**, 1192–1201.

Parlett

Parlett, B.N. (1980), *The Symmetric Eigenvalue Problem*, Prentice–Hall, Inc., Englewood Cliffs, New Jersey.

Pereyra

Pereyra, Victor (1978), PASVA3: An adaptive finite-difference FORTRAN program for first order nonlinear boundary value problems, in *Lecture Notes in Computer Science*, **76**, Springer-Verlag, Berlin, 67–88.

Petro

Petro, R. (1970), Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **13**, 624.

Petzold

Petzold, L.R. (1982), A description of DASSL: A differential/ algebraic system solver, *Proceedings of the IMACS World Congress*, Montreal, Canada.

Piessens et al.

Piessens, R., E. deDoncker-Kapenga, C.W. Uberhuber, and D.K. Kahaner (1983), *QUADPACK*, Springer-Verlag, New York.

Powell

Powell, M.J.D. (1977), Restart procedures for the conjugate gradient method, *Mathematical Programming*, **12**, 241–254.

Powell, M.J.D. (1978), A fast algorithm for nonlinearly constrained optimization calculations, in *Numerical Analysis Proceedings, Dundee 1977, Lecture Notes in Mathematics*, (edited by G.A. Watson), **630**, Springer-Verlag, Berlin, Germany, 144–157.

Powell, M.J.D. (1983), ZQPCVX a FORTRAN *subroutine for convex quadratic programming*, DAMTP Report NA17, Cambridge, England.

Powell, M.J.D. (1985), On the quadratic programming algorithm of Goldfarb and Idnani, *Mathematical Programming Study*, **25**, 46-61.

Powell, M.J.D. (1988), *A tolerant algorithm for linearly constrained optimization calculations*, DAMTP Report NA17, University of Cambridge, England.

Powell, M.J.D. (1989), TOLMIN: *A fortran package for linearly constrained optimization calculations*, DAMTP Report NA2, University of Cambridge, England.

Pruess and Fulton

Pruess, S. and C.T. Fulton (1993), Mathematical Software for Sturm-Liouville Problems, *ACM Transactions on Mathematical Software*, **17**, *3*, 360–376.

Reinsch

Reinsch, Christian H. (1967), Smoothing by spline functions, *Numerische Mathematik*, **10**, 177–183.

Rice

Rice, J.R. (1983), *Numerical Methods, Software, and Analysis*, McGraw-Hill, New York.

Saad and Schultz

Saad, Y., and M.H. Schultz (1986), GMRES: a generalized minimal residual residual algorithm for solving nonsymmetric linear systems, *SIAM J. Sci. Stat. Comput*., **7**, 856–869.

Schittkowski

Schittkowski, K. (1987), *More test examples for nonlinear programming codes*, SpringerVerlag, Berlin, 74.

Schnabel

Schnabel, Robert B. (1985), Finite Difference Derivatives – Theory and Practice, Report, National Bureau of Standards, Boulder, Colorado.

Schreiber and Van Loan

Schreiber, R., and C. Van Loan (1989), A Storage–Efficient *WY* Representation for Products of Householder Transformations, *SIAM J. Sci. Stat. Comp*., Vol. 10, No. 1, pp. 53-57, January (1989).

Scott et al.

Scott, M.R., L.F. Shampine, and G.M. Wing (1969), Invariant Embedding and the Calculation of Eigenvalues for Sturm-Liouville Systems, *Computing*, **4**, 10–23.

Sewell

Sewell, Granville (1982), *IMSL software for differential equations in one space variable*, IMSL Technical Report 8202, IMSL, Houston.

Shampine

Shampine, L.F. (1975), Discrete least-squares polynomial fits, *Communications of the ACM*, **18**, 179−180.

Shampine and Gear

Shampine, L.F. and C.W. Gear (1979), A user's view of solving stiff ordinary differential equations, *SIAM Review*, **21**, 1−17.

Sincovec and Madsen

Sincovec, R.F., and N.K. Madsen (1975), Software for nonlinear partial differential equations, *ACM Transactions on Mathematical Software*, **1**, #3, 232-260.

Singleton

Singleton, R.C. (1969), Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **12**, 185−187.

Smith

Smith, B.T. (1967), *ZERPOL, A Zero Finding Algorithm for Polynomials Using Laguerre's Method*, Department of Computer Science, University of Toronto.

Smith et al.

Smith, B.T., J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y. Ikebe, V.C. Klema, and C.B. Moler (1976), *Matrix Eigensystem Routines – EISPACK Guide*, Springer-Verlag, New York.

Spang

Spang, III, H.A. (1962), A review of minimization techniques for non-linear functions, *SIAM Review*, **4**, 357−359.

Stewart

Stewart, G.W. (1973), *Introduction to Matrix Computations*, Academic Press, New York.

Stewart, G.W. (1976), The economical storage of plane rotations, *Numerische Mathematik*, **25**, 137−139.

Stoer

Stoer, J. (1985), Principles of sequential quadratic programming methods for solving nonlinear programs, in *Computational Mathematical Programming*, (edited by K. Schittkowski), NATO ASI Series, **15**, Springer-Verlag, Berlin, Germany.

Stroud and Secrest

Stroud, A.H., and D.H. Secrest (1963), *Gaussian Quadrature Formulae*, Prentice-Hall, Englewood Cliffs, New Jersey.

Titchmarsh

Titchmarsh, E. *Eigenfunction Expansions Associated with Second Order Differential Equations*, *Part I*, 2d Ed., Oxford University Press, London, 1962.

Trench

Trench, W.F. (1964), An algorithm for the inversion of finite Toeplitz matrices, J*ournal of the Society for Industrial and Applied Mathematics*, **12**, 515 –522.

Walker

Walker, H.F. (1988), Implementation of the GMRES method using Householder transformations, *SIAM J. Sci. Stat. Comput*., **9**, 152–163.

Washizu

Washizu, K. (1968), *Variational Methods in Elasticity and Plasticity*, Pergamon Press, New York.

Watkins and Elsner

Watkins, D.S., and L. Elsner (1990), Convergence of algorithms of decomposition type for the eigenvalue problem, *Linear Algebra and Applications* (to appear).

Weeks

Weeks, W.T. (1966), Numerical inversion of Laplace transforms using Laguerre functions, *J. ACM*, **13**, 419–429.

Wilkinson

Wilkinson, J.H. (1965),*The Algebraic Eigenvalue Problem*, Oxford University Press, London, 635.

Product Support

Contacting Visual Numerics Support

Users within support warranty may contact Visual Numerics regarding the use of the IMSL Libraries. Visual Numerics can consult on the following topics:

- Clarity of documentation
- Possible Visual Numerics-related programming problems
- Choice of IMSL Libraries functions or procedures for a particular problem
- Evolution of the IMSL Libraries

Not included in these consultation topics are mathematical/statistical consulting and debugging of your program.

Consultation

Contact Visual Numerics Product Support by faxing 713/781-9260 or by emailing:

`support@houston.vni.com`.

The following describes the procedure for consultation with Visual Numerics.

1. Include your serial (or license) number
2. Include the product name and version number: IMSL Fortran Library Version 5.0
3. Include compiler and operating system version numbers
4. Include the name of the routine for which assistance is needed and a description of the problem

Index