Lecture 9: Numerical Partial Differential Equations (Part 1)
Finite Difference Method to Solve 2D Diffusion Equation

Consider to solve \( \frac{\partial u}{\partial t} = u_{xx} + u_{yy} + f(x, y) \) in \( \Omega \)
\[
\begin{align*}
\frac{\partial u}{\partial t} &= u_{xx} + u_{yy} + f(x, y) \quad \text{in} \ \Omega \\
u &= 0 \quad \text{on} \ \partial \Omega
\end{align*}
\]
by using an forward in time and backward in space (FTCS or explicit) finite difference scheme.

Here \( \Omega = [0, a] \times [0, b] \), \( f(x, y) = xy \). \( a \) and \( b \) are constants and \( a \geq 0 \).
Finite Differences

• Spatial Discretization: \( 0 = x_0 < \cdots < x_M = a \) with \( x_i = \frac{i}{M} a \) and \( 0 = y_0 < \cdots < y_N = b \) with \( y_j = \frac{j}{N} b \). Define \( \Delta x = \frac{a}{M} \) and \( \Delta y = \frac{b}{N} \).

• Differential quotient:

\[
\begin{align*}
\frac{\partial^2 u}{\partial x^2}(x_i, y_j, t) &\sim \frac{u(x_{i-1}, y_j, t) - 2u(x_i, y_j, t) + u(x_{i+1}, y_j, t)}{\Delta x^2} \\
\frac{\partial^2 u}{\partial y^2}(x_i, y_j, t) &\sim \frac{u(x_i, y_{j-1}, t) - 2u(x_i, y_j, t) + u(x_i, y_{j+1}, t)}{\Delta y^2} \\
\frac{\partial u}{\partial t}(x_i, y_j, t_n) &\sim \frac{u(x_i, y_j, t_{n+1}) - u(x_i, y_j, t_n)}{\Delta t}
\end{align*}
\]
Insert quotients into PDE yields:

\[
v(x_i, y_j, t_{n+1}) = v(x_i, y_j, t_n) + \Delta t \left( \frac{v(x_{i-1}, y_j, t_n) - 2v(x_i, y_j, t_n) + v(x_{i+1}, y_j, t_n)}{\Delta x^2} + \frac{v(x_i, y_{j-1}, t_n) - 2v(x_i, y_j, t_n) + v(x_i, y_{j+1}, t_n)}{\Delta y^2} \right) + \Delta tf(x_i, y_j)
\]

Or in short notation

\[
v_{i,j}^{n+1} = v_{i,j}^n + \Delta t \left( \frac{v_{i-1,j}^n - 2v_{i,j}^n + v_{i+1,j}^n}{\Delta x^2} + \frac{v_{i,j-1}^n - 2v_{i,j}^n + v_{i,j+1}^n}{\Delta y^2} \right) + \Delta tf(x_i, y_j)
\]

Boundary conditions:

\[
v_{0,j}^{n+1} = 0; \quad v_{M,j}^{n+1} = 0; \quad v_{i,0}^{n+1} = 0; \quad v_{i,N}^{n+1} = 0.
\]
Parallel Computation with Grids

- Partition solution domain into subdomains.
- Distribute subdomains across processors
- Communication between processors is needed to provide interface between subdomains.
  - Communication is needed when stencil for given grid point includes points on another processor
  - For efficiency, ghost points are used for message passing at the end (or begin) of each iteration. Ghost points overlap between two subdomains, so as subgrids.
Ghost Points

grid points  ghost points  stencil
Grid Structure

struct _RECT_GRID {
    double L[3];    /* Lower corner of rectangle containing grid */
    double U[3];    /* Upper corner of rectangle containing grid */
    double h[3];    /* Average grid spacings in the grid */
    int   gmax[3];  /* Number of grid blocks */
    int   dim;      /* Dimension of Grid */

    /* Specifications for virtual domains and variable grids */

    double GL[3];   /* Lower corner of global grid */
    double GU[3];   /* Upper corner of global grid */
    double VL[3];   /* Lower corner of virtual domain */
    double VU[3];   /* Upper corner of virtual domain */
    int    lbuf[3]; /* Lower buffer zone width */
    int    ubuf[3]; /* Upper buffer zone width */
};

typedef struct _RECT_GRID RECT_GRID;
Solution Storage

#define soln(u, ic, gr) (u[n_indx((ic), (gr))])
#define n_indx(ic, gr) ((ic)[1]*(gr)->gmax[0]+(gr)->lbuf[0]+(gr)->ubuf[0]) + (ic)[0])

double *u_store, *u;
int x_size, y_size, i, ic[2];
RECT_GRID *gr;
....
// properly initialize gr.
....
x_size = gr->gmax[0]+gr->lbuf[0]+gr->ubuf[0];

u_store = new double [x-size*y-size];
u = u_store + gr->lbuf[1]*x_size + gr->lbuf[0];

// show state at the first row of the grid
ic[1] = 0;
for(i = -gr->lbuf[0]; i < gr->lbuf[0]+gr->ubuf[0]; i++)
{
  ic[0] = i;
  cout << "state = " << soln(u,ic,gr) << endl;
}
Communication of Rectangular Lattice States

\[
\begin{align*}
\text{i} &= 0 \\
\text{l}[0] + lbuf[0] & \quad u[0] - ubuf[0] \\
\text{l}[1] & \quad \text{l}[0] + lbuf[0] \quad u[0] - ubuf[0] \\
\text{R} & \quad \text{E} \quad \text{C} \quad \text{S} \quad \text{E} \quad \text{I} \quad \text{N} \quad \text{V} \quad \text{D} \quad \text{E} \\
\text{E} & \quad \text{E} \quad \text{S} \quad \text{C} \quad \text{E} \quad \text{E} \quad \text{N} \quad \text{D} \quad \text{V} \quad \text{E} \\
\text{I} & \quad \text{N} \quad \text{I} \quad \text{N} \quad \text{I} \\
\text{V} & \quad \text{D} \quad \text{V} \quad \text{D} \quad \text{V} \\
\text{E} & \quad \text{E} \quad \text{E} \quad \text{E} \quad \text{E} \\
\text{Vl}[0] & \quad \text{l}[0] \quad \text{u}[0] \quad \text{Vu}[0]
\end{align*}
\]
\[ i = 1 \]

\[ \text{Vu}[1] \]
\[
\begin{array}{c}
\text{SEND} \\
\text{RECEIVE} \\
\end{array}
\]

\[ \text{I}[1] \]
\[
\begin{array}{c}
\text{SEND} \\
\text{RECEIVE} \\
\end{array}
\]

\[ \text{VI}[1] \]
\[
\begin{array}{c}
\text{RECEIVE} \\
\end{array}
\]

\[ \text{VI}[0] \text{ Vu}[0] \]
// Assume we have created a Cartesian grid topology with communicator
// grid_comm

void scatter_states(
    double       *u,
    RECT_GRID    *gr)
{
    int my_id, side, dim = 2, i;
    int me[2];

    MPI_Comm_rank(grid_comm, &my_id);
    MPI_Cart_coords(grid_comm, my_id, 2, me);
    for(i = 0; i < dim; i++)
    {
        for(side = 0; side < 2; side++)
        {
            MPI_Barrier(MPI_Comm);
            pp_send_interior_states(me, i, side, u);
            pp_receive_interior_states(me, i, (side+1)%2, u);
        }
    }
}
void pp_send_interior_states(
    int *me,
    int dir,
    int side,
    double *u)
{
    int him[2], i, dim = 2;
    int dst_id;
    int L[3], U[3];
    double *storage;

    for (i = 0; i < dim; i++)
        him[i] = me[i];
    him[dir] = (me[dir] + 2*side - 1);
    if (him[dir] < 0)
        him[dir] = G[dir] - 1;
    if (him[dir] >= G[dir])
        him[dir] = 0;
    MPI_Cart_rank(grid_comm, him, &dst_id);

    // figure out region in which the data need to be sent
    set_send_domain(L,U,dir,side,gr);

    storage = new double [(U[0]-L[0])*(U[1]-L[1])];
    // collect data and put into storage
    ... // ... 
    MPI_Bsend(storage, (U[0]-L[0])*(U[1]-L[1]), MPI_DOUBLE, dst_id, 100, MPI_COMM);
}
set_send_domain(int *L, int *U, int dir, int side, RECT_GRID *gr)
{
    int dim = gr->dim;
    int *lbuf = gr->lbuf;
    int *ubuf = gr->ubuf;
    int *gmax = gr->gmax;
    int j;

    for (j = 0; j < dir; ++j)
    {
        L[j] = -lbuf[j];
        U[j] = gmax[j] + ubuf[j];
    }

    if (side == 0)
    {
        L[dir] = 0;
        U[dir] = lbuf[dir];
    }
    else
    {
        L[dir] = gmax[dir] - ubuf[dir];
        U[dir] = gmax[dir];
    }

    for (j = dir + 1; j < dim; ++j)
    {
        L[j] = -lbuf[j];
        U[j] = gmax[j] + ubuf[j];
    }
}
void pp_receive_interior_states(
    int *me,
    int dir,
    int side,
    double *u)
{
    int him[2], i, dim = 2;
    int src_id;
    int L[3], U[3];
    double *storage;
    MPI_Status *status;

    for (i = 0; i < dim; i++)
        him[i] = me[i];
    him[dir] = (me[dir] + 2*side - 1);
    if (him[dir] < 0)
        him[dir] = G[dir] - 1;
    if (him[dir] >= G[dir])
        him[dir] = 0;
    MPI_Cart_rank(grid_comm, him, &src_id);

    //// figure out region in which the data need to be sent
    set_receive_domain(L,U,dir,side,gr);

    storage = new double [(U[0]-L[0])*(U[1]-L[1])];

    MPI_Recv(storage, (U[0]-L[0])*(U[1]-L[1]), MPI_DOUBLE, src_id, 100, MPI_COMM,&status);

    // Put received data into proper places of u
}
```c
set_receive_domain(int *L, int *U, int dir, int side, RECT_GRID *gr) {
    int dim = gr->dim;
    int *lbuf = gr->lbuf;
    int *ubuf = gr->ubuf;
    int *gmax = gr->gmax;
    int j;
    for (j = 0; j < dir; ++j) {
        L[j] = -lbuf[j];
        U[j] = gmax[j] + ubuf[j];
    }
    if (side == 0) {
        L[dir] = -lbuf[dir];
        U[dir] = 0;
    } else {
        L[dir] = gmax[dir];
        U[dir] = gmax[dir] + ubuf[dir];
    }
    for (j = dir + 1; j < dim; ++j) {
        L[j] = -lbuf[j];
        U[j] = gmax[j] + ubuf[j];
    }
}
```
int main()
{
    int i, j, k, Max_steps = 10000, ic[2];
    RECT_GRID *gr;
    double *u, *u_prev, *tmp;

    // initialize lattice grid: gr
    // initialize storage: *u;
    // initialize state
    // computation
    for(i = 0; i < Max_steps; i++)
    {
        // time stepping
        for(j = 0; j < gr->gmax[0]; j++)
        {
            ic[0] = j;
            for(k = 0; k < gr->gmax[1]; k++)
            {
                ic[1] = k;
                // update soln: soln(u, ic, gr) = soln(u_prev, ic, gr) + ... ;
            }
        }
        // communication to update ghost points
        scatter_states(u, gr);

        // swap storage for next step
        tmp = u;     u = u_prev;
        u_prev = tmp;
    }
}
Lecture 9: Numerical Partial Differential Equations (Part 2)
Finite Difference Method to Solve Poisson’s Equation

- Poisson’s equation in 1D:
  \[
  \begin{cases}
  -\frac{d^2u}{dx^2} = f(x), \quad x \in (0,1) \\
  u(0) = u(1) = 0
  \end{cases}
  \]

- Spatial Discretization: \(0 = x_0 < \cdots < x_M = 1\).
  Define \(\Delta x = \frac{1}{M}\). Then \(x_i = i\Delta x\).

- \(\frac{d^2u(x_i)}{dx^2} \sim \frac{u(x_{i-1}) - 2u(x_i) + u(x_{i+1})}{\Delta x^2}\)

- Stencil of finite difference approximation

\[\begin{array}{c}
1 \\
\hline
-2 \\
1
\end{array}\]
Finite difference equations: for \( i = 1, \ldots, M - 1 \)
\[-u_{i-1} + 2u_i - u_{i+1} = \Delta x^2 f_i\]
\[u_0 = 0\]
\[u_M = 0\]

with \( f_i = f(x_i) \)

Put into matrix equation format:
Let \( \mathbf{u} = (u_1, u_2, \ldots, u_{M-1})^T, \mathbf{f} = (f_1, f_2, \ldots, f_{M-1})^T \)
\[A\mathbf{u} = \Delta x^2 \mathbf{f}\]

\[A = \begin{bmatrix}
2 & -1 & & & \\
-1 & 2 & -1 & & \\
& -1 & 2 & -1 & \\
& & \ddots & \ddots & \ddots \\
& & & -1 & 2 & -1 \\
& & & & -1 & 2 & -1 \\
& & & & & -1 & 2
\end{bmatrix}\]
2D Poisson’s Equation

Consider to solve

\[
-\left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = f(x, y), \quad (x, y) \in \Omega
\]

\[
u(x, y) = 0 \quad \text{on} \quad \partial \Omega
\]

with \( \Omega \) is rectangle \((0,1) \times (0,1) \) and \( \partial \Omega \) is its boundary.

• Define \( h = \frac{1}{M} \).

• Spatial Discretization: \( 0 = x_0 < \cdots < x_M = a \) with \( x_i = ih \) and \( 0 = y_0 < \cdots < y_M = 1 \) with \( y_j = jh \).
Finite difference equation at grid point \((i, j)\):

\[
- \left( \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} \right) = f(x_i, y_j) \quad \text{or} \\
- u_{i,j-1} - u_{i-1,j} + 4u_{i,j} - u_{i+1,j} - u_{i,j+1} = h^2 f(x_i, y_j)
\]

- Five-point stencil of the finite difference approximation
\[ u_{ij}^{(k+1)} = (1 - w)u_{ij}^{(k)} \]
\[ + \frac{w}{4} \left( h^2 f_{ij} + u_{i-1,j}^{(k+1)} + u_{i,j-1}^{(k+1)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)} \right) \]

This is completely sequential.
Red-Black Ordering

• Color the alternate grid points in each dimension red or black
R/B SOR

- First iterates on red points by
  \[ u_{ij}^{(k+1)} = (1 - w)u_{ij}^{(k)} + \frac{w}{4} \left( h^2 f_{ij} + u_{i-1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)} \right) \]

- Then iterates on black points by
  \[ u_{ij}^{(k+1)} = (1 - w)u_{ij}^{(k)} + \frac{w}{4} \left( h^2 f_{ij} + u_{i-1,j}^{(k+1)} + u_{i,j-1}^{(k+1)} + u_{i+1,j}^{(k+1)} + u_{i,j+1}^{(k+1)} \right) \]

- R/B SOR can be implemented in parallel on the same color grid points.

- The renumbering of the matrix \( A \) changes the iteration formula.
For the example just shown:

$$A = \begin{bmatrix} D_r & -C \\ -C^T & D_b \end{bmatrix}$$

Diagonal matrices $D_r = D_b = 4I_8$. 
• Using GS:

\[
\begin{bmatrix}
D_r & 0 \\
-C^T & D_b
\end{bmatrix}
\begin{bmatrix}
\mathbf{u}_r^{(k+1)} \\
\mathbf{u}_b^{(k+1)}
\end{bmatrix}
= 
\begin{bmatrix}
0 & 0 \\
0 & 0
\end{bmatrix}
\begin{bmatrix}
\mathbf{u}_r^{(k)} \\
\mathbf{u}_b^{(k)}
\end{bmatrix}
+ h^2 \mathbf{f}
\]

Here \( \mathbf{u}_r = (u_1, u_2, u_3, u_4, \ldots, u_8)^T \)

\( \mathbf{u}_b = (u_9, u_{10}, u_{11}, u_{12}, \ldots, u_{16})^T \)
Parallel R/B SOR

P0

1 9 2

P2

15 7 16

5 13 6

11 3 12

4

P3

P1

8

14
Algorithm

While error > TOL, do:

- Compute all red-points
- Send/Recv values of the red-points at the boarder of the subdomain to neighboring processes
- Compute all black-points
- Send/Recv values of the black-points at the boarder of the subdomain to neighboring processes

Compute residual error

Endwhile
References


Lecture 9: Numerical Partial Differential Equations (Part 3) – MPI user-defined Datatype
From SOR.c

```c
ierr = MPI_Sendrecv(&Phi[(i_send*ColNumber)+j_send], 1, MPI_InteriorPointsCol, mynode.west, tag, &Phi[(i_recv*ColNumber)+j_recv], 1, MPI_InteriorPointsCol, mynode.east, tag, MPI_COMM_WORLD, &status);
```

Derived Datatypes

- Techniques for describing non-contiguous and heterogeneous (structure) data
  - Data are not contiguous in memory
- MPI uses *derived datatypes* for this purpose.
MPI type-definition functions

- `MPI_Type_Contiguous`: a replication of datatype into contiguous locations
- `MPI_Type_vector`: replication of datatype into locations that consist of equally spaced blocks
- `MPI_Type_commit`: commit user defined derived datatype
- `MPI_Type_free`: free the derived datatype
- `MPI_Type_create_hvector`: like vector, but successive blocks are not multiple of base type extent
- `MPI_Type_indexed`: non-contiguous data layout where displacements between successive blocks need not be equal
- `MPI_Type_create_struct`: most general – each block may consist of replications of different datatypes
• **MPI_Type_contiguous** (int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
  – IN count (replication count)
  – IN oldtype (base data type)
  – OUT newtype (handle to new data type)

• Creates a new type which is simply a replication of oldtype into contiguous locations

Example 1:
/* create a type which describes a line of ghost cells */
/* buf[0], buf1],...,buf[nxl-1] set to ghost cells */
    int nxl;
    MPI_Datatype ghosts;

    **MPI_Type_contiguous** (nxl, MPI_DOUBLE, &ghosts);
    **MPI_Type_commit**(ghosts)
    **MPI_Send** (buf, 1, ghosts, destin, tag, MPI_COMM_WORLD);
    ..
    ..
    **MPI_Type_free**(ghosts);

From red_black_SOR.c
ierr = **MPI_Type_contiguous**(BlockLenH,MPI_DOUBLE,&MPI_InteriorPointsRow);
count = 4;
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);

<table>
<thead>
<tr>
<th></th>
<th>1.0</th>
<th>2.0</th>
<th>3.0</th>
<th>4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>6.0</td>
<td>7.0</td>
<td>8.0</td>
<td></td>
</tr>
<tr>
<td>9.0</td>
<td>10.0</td>
<td>11.0</td>
<td>12.0</td>
<td></td>
</tr>
<tr>
<td>13.0</td>
<td>14.0</td>
<td>15.0</td>
<td>16.0</td>
<td></td>
</tr>
</tbody>
</table>

\[a[4][4]\]

MPI_Send(&a[2][0], 1, rowtype, dest, tag, comm);

<table>
<thead>
<tr>
<th></th>
<th>9.0</th>
<th>10.0</th>
<th>11.0</th>
<th>12.0</th>
</tr>
</thead>
</table>

1 element of rowtype
• **MPI_Type_vector** (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
  – IN count (number of blocks)
  – IN blocklength (number of elements per block)
  – IN stride (spacing between start of each block, measured in # elements)
  – IN oldtype (base datatype)
  – OUT newtype (handle to new type)

  – Allows replication of old type into locations of equally spaced blocks. Each block consists of same number of copies of oldtype with a stride that is multiple of extent of old type.

```c
MPI_Type_vector(count, blocklength, stride, MPI_FLOAT, &columntype);
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2.0</td>
<td>3.0</td>
<td>4.0</td>
</tr>
<tr>
<td>5.0</td>
<td>6.0</td>
<td>7.0</td>
<td>8.0</td>
</tr>
<tr>
<td>9.0</td>
<td>10.0</td>
<td>11.0</td>
<td>12.0</td>
</tr>
<tr>
<td>13.0</td>
<td>14.0</td>
<td>15.0</td>
<td>16.0</td>
</tr>
</tbody>
</table>

```c
MPI_Send(&a[0][1], 1, columntype, dest, tag, comm);
```
Example 1. Use MPI_Type_vector to send a submatrix: Suppose there is a 4x4 matrix and we want to send the middle four elements.
Type_vector.c

Example 2. red_black_SOR.c

```c
ierr = MPI_Type_vector(BlockLenV,1,ColNumber,MPI_DOUBLE,
                        &MPI_InteriorPointsCol);
```

• **MPI_Type_create_hvector** (int count, int blocklength, MPI_Aint stride, MPI_Datatype old, MPI_Datatype *new)
  
  – IN count (number of blocks)
  – IN blocklength (number of elements/block)
  – IN stride (number of bytes between start of each block)
  – IN old (old datatype)
  – OUT new (new datatype)

• Same as MPI_Type_vector, except that stride is given in bytes rather than in elements (‘h’ stands for ‘heterogeneous’).
• **MPI_Type_indexed** (int count, int *array_of_blocklengths, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype);
  – IN count (number of blocks)
  – IN array_of_blocklengths (number of elements/block)
  – IN array_of_displacements (displacement for each block, measured as number of elements)
  – IN oldtype
  – OUT newtype

• Displacements between successive blocks need not be equal. This allows gathering of arbitrary entries from an array and sending them in a single message.

• The high level view of MPI_Type_indexed is that you create a type to represent a particular part of a matrix. And then you commit that type using MPI_Type_commit. So, You specify an arbitrary number of blocks(i.e. continuous elements) of arbitrary lengths within the matrix. **Count** is the total number of elements that you are sending of **old_type**, **array_of_blocklengths** is an array the length of each block you will send, and **array_of_displacements** is an array of where each block begins (i.e. displacement from the beginning of the array.
• **Example.** Use `MPI_Type_indexed` to send a subarray: `sample_Type_index.c`

```c
count = 2;  blocklengths[0] = 4;
displacements[0] = 5;  blocklengths[1] = 2;
displacements[1] = 12;

MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);

MPI_Send(&a, 1, indextype, dest, tag, comm);
```

1 element of indextype