Bertini User's Manual

Dan Bates,
with contributions from Jon Hauenstein

December 16, 2008

Contents

Chapter 1

Introduction to Bertini

Numerical algebraic geometry is a young and burgeoning field of mathematical research. It lies at the intersection of algebraic geometry and numerical analysis and addresses problems arising both in the sciences and engineering and also in pure algebraic geometry. Bertini is a software package for computation in numerical algebraic geometry.

1.1 What Bertini does

Generally speaking, numerical algebraic geometry provides algorithms for computing and manipulating structures of an algebro-geometric nature. A number of these algorithms have been implemented in Bertini (and more are on the way). For example, Bertini can be used to produce all isolated solutions of a system of polynomials with complex coefficients. These points can be computed with up to several hundred digits of accuracy since Bertini makes use of multiple precision Even better, the user does not need to know *a priori* the level of precision necessary to attain the desired accuracy because Bertini may be told to use adaptive precision changes. Details about such zero-dimensional solving may be found in Chapter 4.

Another of Bertini's key capabilities is that it can find "witness points" on every irreducible component of the algebraic set corresponding to a system of polynomials with complex coefficients. [28] is a good reference for the definitions, theory, and algorithms behind such "witness sets." The bottom line is that Bertini will detect the irreducible decomposition of the algebraic set by specifying at least one point on each component. Once such a witness set is computed, the user may sample any component (i.e., produce as many points on the component as desired) and perform component membership (i.e., specify a list of points and ask Bertini to determine whether any of the points lie on the algebraic set correspoding to the witness set).

Bertini allows the user to choose from several different types of runs and to specify a large number of tolerances (although all tolerances come with reasonable default settings). The point of this manual is to indicate how to use Bertini (see §1.4) – not the theory behind the algorithms. If you are new to numerical algebraic geometry or would like a refresher on some of the latest algorithms, here is a list of some of the more recent references (mostly listed by date of publication):

- General references: [18], [26], [28]

- Finding isolated solutions: [20], [1], [18], [26], [28]
- Finding witness sets: [24], [26], [28]
- Endgames: [21], [22], [23]
- Sophisticated start systems: [18]
- Adaptive multiple precision: [4], [5]
- Deflation: [17]
- Intersecting solution sets: [25]
- Equation by equation solving: [27], [16]
- Applications within mathematics: [10], [6], [19], [7], [8]
- Applications in engineering and the sciences[1]: [34], [35], [28], [2], [36], [32]
- Related software: [33], [11], [37], [12], [15], [30], [39]

1.2 Where Bertini may be found

The official Bertini website is http://www.nd.edu/∼sommese/bertini. There you will find the current version of Bertini, information regarding bug fixes and revisions, a form for reporting bugs and providing feedback, example input files, and license information. Scripts written in Maple, Matlab, and/or C for creating input files and interpreting output files are or will be posted on the website of Daniel Bates. Errors and suggested changes in this manual may also be reported on the Bertini website via the feedback form.

The developers of Bertini are aware of the many good reasons for going open source, but only the executable is presently available to the public. To download a copy of Bertini, just go to the website, find the version of the executable that best matches your computer, and click the link to download it. This will get you a `.tar.gz` file of a directory. To unpack the directory, just type `tar -zxvf FILE_NAME` at the command line in the directory in which you saved the file (where `FILE_NAME` is the name of the file you downloaded, e.g., `BertiniLinux32_beta.tar.gz`). The directory that is created will contain a copy of this user's manual, the executable, a directory containing several examples to get you started, and a README file explaining the examples.

Cygwin users should already have GMP and MPFR installed. If they are not, there are directions for adding modules to Cygwin on the Cygwin webpage. It is quite easy. Other Linux users do not need to worry about installing GMP, MPFR, or any other packages as we were able to make the links to the appropriate libraries static in that case.

If no version of Bertini works on your machine, please contact the development team as described on the website. Please note that Bertini is designed for use with Linux/Unix or emulators

[1] Jan Verschelde contains a lengthy list of polynomial systems coming from applications on his website, including references to their original appearance in the literature.

such as Cygwin on Windows. It might be possible to produce versions that can be used on other operating systems, but we can make no such guarantees.

Some users have succeeded in using the Linux version on Macs using OS X. Plans are in place to create a version for earlier Mac operating systems, in the near future. Bertini has also been used on Windows machines without Cygwin, by including appropriate dll's in the directory containing the Bertini.exe file. However, such usage has known problems and has not yet been fully tested. Thus, Bertini is only supported on Windows machines if Cygwin is used.

Details about how to use Bertini once it is downloaded are given in §2.1.2.

1.3 Who is developing Bertini

Bertini is under ongoing development by the development team, which consists of Daniel J. Bates (Colorado State University), Jonathan D. Hauenstein (University of Notre Dame), Andrew J. Sommese (University of Notre Dame), and Charles W. Wampler II (General Motors Research and Development).

1.4 How to use this manual

As mentioned above, this manual is designed to provide users with the knowledge necessary to use Bertini. For background on numerical algebraic geometry, please refer to the references listed in §1.1. The best way to learn how to use Bertini is through examples, so this manual consists mostly of examples, syntax rules, and configuration options. In particular, Chapter 2 is key for first-time users in that it provides very basic examples and a description of how to set up and carry out basic runs of various types. Chapter 3 is intended mostly for reference and should be useful (especially §3.2) when Bertini will not run due to a syntax error in the input file. Chapters 4 and 5 give further details about and options for zero- and positive-dimensional runs, respectively. Algorithms and output are described in those two chapters, and details regarding the special case of user-defined homotopies are given in §2.2.3.

A few details regarding the parallel versions of Bertini are provided in Chapter 6 and the inside workings of Bertini are provided in Chapter 7. [2] Finally, the appendix gives a detailed list of all user-changeable options.

The best way to learn how to use Bertini is to try the simple problems included with the distribution, using this manual as your guide, as described above. Daniel Bates gave a talk in October 2006 about using Bertini, around the time the beta version was first released. This talk took place at the IMA (Institute for Mathematics and its Applications) in Minneapolis, MN, USA. It was videotaped and is available to the public by navigating through the IMA website (http://www.ima.umn.edu – look for a link to talk materials, then the 2006 Workshop on Software for Algebraic Geometry).

[2] Please note that Chapter 7 is not maintained as frequently as the rest of this manual, so a fair bit of the information in that chapter is actually out of date.

1.5 Acknowledgements

The development of Bertini has been supported generously by a number of sources, including the Duncan Chair at the University of Notre Dame, the University of Notre Dame, National Science Foundation grants 0410047 and 0105653, General Motors, the Institute for Mathematics and its Applications, the Arthur J. Schmitt Foundation, and the University of Notre Dame Center for Applied Mathematics.

Chapter 2

Getting started

2.1 Basics of a Bertini run

Bertini is not yet entirely interactive. Most of the input and most of the output may be found in files. After providing the input files (described in the next subsection and in detail in the next chapter), the user calls Bertini (as described in §2.1.2) and can find the output on the screen and in files, as described in §4.2 and §4.3 for zero-dimensional runs and in §5.2 and §5.3 for positive-dimensional runs. Special techniques, such as user-defined homotopies, component sampling, and component membership testing, endpoint sharpening, and deflation require a little extra work, so they are treated separately.

2.1.1 How to provide input

For most types of runs, Bertini needs only one file, the `input` file. The `input` file consists of an optional list of configurations (e.g., the desired accuracy of the solutions), declarations for almost all names used in the definition of the polynomials (e.g., all variables), and the target polynomial system itself. For example, the following simple `input` file is appropriate for finding the isolated complex solutions of $x^2 - 1$ (with no special configurations):

```
function f;
variable_group x;
f = x^2 - 1;
END;
```

Note that each line must end with a semicolon and that the entire file is terminated with the keyword `END`. All white space (spaces, tabs, and blank lines) is ignored by Bertini. Since no configurations are listed, all default settings for the user-changeable tolerances of Appendix A will be used. Suppose, though, that you would like to track to only 8 digits of accuracy, rather than the default. Then the `input` file would have the form:

```
CONFIG
```

```
FINALTOL: 1e-8;
END;
INPUT
function f;
variable_group x;
f = x^2 - 1;
END;
```

In this case, the beginning of the input section was specified by including the keyword `INPUT` just before the declarations. This was necessary because of the inclusion of the configuration section at the top of the file, starting with the keyword `CONFIG` and ending with `END;`. The configurations are just listed one per line between those keywords. More details about the `input` file can be found in the examples later in this chapter, and the exact syntax is given in the next chapter. It should be noted, though, that the `input` file need not be named *input*; any legal file name will suffice. The name `input` is used here for convenience.

In almost all situations, Bertini will automatically create a homotopy, including a start system, and solve the start system to find the points at the beginning of each path. The only exceptional case is when the user provides his or her own homotopy. It is impossible in that case for Bertini to find the solutions of the start system (without homotoping to it from yet another system). Thus, the user must provide a set of starting points that solve the start system in the case of a user-defined homotopy. These points must be provided in a file named `start` the syntax of which is given in §3.3. It should be noted, though, that as opposed to the `input` file, the `start` file must actually be named *start*.

The `member_points` file is needed in addition to the `input` file in the case of component membership testing. This file contains a list of all points to be tested, is identical in structure to the `start` file, and must be named `member_points`.

One more input file of interest that is automatically generated as output during a run is the `witness_data` file. This is a terse form of the witness data of an algebraic set (i.e., the witness points and the linear slices). Although this file could technically be produced manually, the best way to create this file is by executing a positive-dimensional Bertini run (in which case the `witness_data` file is produced automatically). Since the syntax of the `witness_data` file is so complicated and has so many cases, the exact syntax will not be provided in this version of the manual. As with the `start` file, the `witness_data` file must have exactly that name, and this file must be specified in order to use either component sampling or component membership testing.

2.1.2 How to run Bertini

Once all appropriate input files have been created (typically just `input`), running Bertini is simple. From the command line, in the directory containing the executable, just type `./bertini FILE_NAME`, where `FILE_NAME` is either the name of a file in the same directory or the path to a file in another directory (including the file name itself). If no file is specified, Bertini will use as input the file named `input` in the current working directory. If no file is specified and `input` does not exist, Bertini will exit after printing an error message to the screen.

One can specify the path to the executable as well, so Bertini can be run in any directory to which the user has access. Even better, by placing the executable in a `bin` directory (and perhaps logging out and then back into your account), Bertini can be called from anywhere in the file tree just by typing `bertini` (without a path). In any case, all output files are created in the directory from which the call is made (i.e., the current working directory). Of course, the user must have write priveleges in that directory. Please note that when using Cygwin, the executable may not actually be named `bertini` but rather `bertini.exe`.

Suppose the directory in which the executable resides has the path `~/bertini_dir/`, the input file of interest (named `my_input`) is in `~/input_files/`, and the current directory is `~/elsewhere/`. Then, the command `~/bertini_dir/bertini ~/input_files/my_input` will run the polynomial system of interest and create all output files in `~/elsewhere/`.

2.2 Some first examples

2.2.1 Zero-dimensional examples

Let us first revisit the example

```
function f;
variable_group x;
f = x^2 - 1;
END;
```

from §2.1.1.

Note that the keyword for the variables in the above example is `variable_group` and not just "variable." This is how Bertini knows to homogenize the input system but provide output in complex space rather than projective space. Alternatively, one may choose to provide homogenized input, in which case the output is also given in projective space (although no points at infinity are identified, since there is no canonical sense of infinity). The keyword for variables in that case is `hom_variable_group`. The input file for a 1-homogenized form of $x^2 - 1$ (where "1" indicates that we are working in a single projective space rather than a product) is then

```
function f;
hom_variable_group x, y;
f = x^2 - y^2;
END;
```

This homogenized input also shows an advantage of using Bertini: it can handle nonsquare systems, i.e., systems for which the number of variables and the number of equations differ. In fact, before any tracking is carried out, Bertini first computes the rank of the target system to check whether there could be any isolated solutions. If so, it squares the functions up to the number of variables (which is reasonable by Bertini's Theorem). If not, Bertini immediately reports that there are no isolated solutions and exits (in the case of zero-dimensional runs).

Bertini can also handle systems defined over a product of complex or projective spaces. In particular, if there are m terms in the product, the user must specify m variable groups, in the form of m lines in the input file, each designated by the appropriate keyword. Please note that projective and complex spaces cannot be mixed, i.e., the user must choose between using `hom_variable_group` and using `variable_group` throughout a single `input` file. If the user chooses to provide a homogenized target system, Bertini will automatically check the homogenization and display an error message and quit in case of an error. Here is an example of an `input` file for a 2-homogenized target system (adapted from [14]):

```
variable_group z1;
variable_group z2;
function f1, f2;
f1 = (29/16)*z1^3 - 2*z1*z2;
f2 = z2 - z1^2;
END;
```

Now suppose you want Bertini to follow the paths of this example very closely and that you want the final solutions correct to 14 digits. Then the `input` file would be:

```
CONFIG
TRACKTOLBEFOREEG: 1e-8;
TRACKTOLDURINGEG: 1e-11;
FINALTOL: 1e-14;
END;
INPUT;
variable_group z1;
variable_group z2;
function f1, f2;
f1 = (29/16)*z1^3 - 2*z1*z2;
f2 = z2 - z1^2;
END;
```

Suppose you then decide that you want even more accuracy - 20 digits - but you know that Bertini uses only 16 digits of precision as a default. You might then want to increase the precision used by Bertini, say to 128 bits (or about 38 digits). To do so, you could just add the lines

```
MPTYPE: 1;
PRECISION: 128;
```

anywhere in the `CONFIG` section of the `input` file. Realistically, you don't really want to waste your time with guessing how much precision is adequate since you would likely choose either too little precision (and need to try again after waiting through the run) or too much precision (causing an unnecessary slowdown in tracking). Instead, just let Bertini change precision on the fly whenever needed (i.e., use adaptive precision). Instead of the previous two lines, then, you would need to add at least one[1]:

```
MPTYPE: 2;
```

Bertini also allows for the use of subfunctions when defining functions. This capability was built in because many systems coming from engineering have symmetry, so lumping many identical parts of the input file together into one subfunction results in increased efficiency in function evaluation. Function and Jacobian evaluation are the most expensive parts of almost all runs. Note that subfunctions are the only names in an `input` file that do not need to be declared. Here is an example in which the subfunction appears in every function (adapted from [28]):

```
variable_group x, y, z;
function f1, f2, f3;
S = x^2+y^2+z^2-1;
f1 = (y-x^2)*S*(x-0.5);
f2 = (z-x^3)*S*(y-0.5);
f3 = (y-x^2)*(z-x^3)*S*(z-0.5);
END;
```

If you are reading through this manual from beginning to end, you should now be prepared to run most zero-dimensional examples just by using the format of the files in this section (and perhaps referring to the syntax rules provided in the next chapter and the table of configurations provided in Appendix A).

2.2.2 Positive-dimensional examples

The `input` files for positive-dimensional runs are almost identical to those for zero-dimensional runs. In fact, other than a few configuration settings, it is impossible to distinguish between the two. Here are the differences:

- To specify that Bertini should do a positive-dimesional run (rather than the default zero-dimensional run), the user must change the `TRACKTYPE` in the `CONFIG` portion of the input file. For example, `TRACKTYPE: 1;` specifies a basic positive-dimensional run, 2 specifies sampling, and 3 specifies membership testing.
- The user may specify only one variable group, i.e., the system may be defined over either a single complex space or a single projective space (no products).

[1]Note that other configurations should be changed as well in order to maximize either the speed or the security of your run. Please refer to the description of MPTYPE given in Appendix A for details

Earlier versions of Bertini only allowed for the use of regular precision in positive dimensions, but the positive-dimensional algorithms have now been implemented in both fixed and adaptive multiple precision.

For example, a witness set for the twisted cubic curve could be computed by Bertini using the following `input` file:

```
CONFIG
TRACKTYPE: 1;
END;
INPUT
variable_group x, y, z;
function f1, f2;
f1 = x^2-y;
f2 = x^3-z;
END;
```

Using this `input` file, Bertini will produce three points on the twisted cubic curve. Suppose you want a few more points, say 200. Then, simply change the `TRACKTYPE` line to

```
TRACKTYPE: 2;
```

Upon calling Bertini (assuming the `witness_data` file from the previous run is still intact), a catalog of all components in all dimensions will appear on the screen. Bertini will prompt you to enter a codimension, a component number, and a number of points (to which you would respond 2, 1, and 200, respectively). The final prompt asks where to print the sample points: enter 0 to print the points to a file called `sample_points` and enter 1 to print the points to the screen. Bertini would then simply move the slices around and report all 200 desired points.

Finally, suppose you have a point in $\mathbb{C}^3$ that you think might be on the twisted cubic curve. You could place that point into a file named `member_points` (with syntax identical to the `start` file described in §3.3), change `TRACKTYPE` to 3, and again ensure that the `witness_data` file is intact. Bertini would then perform the component membership test on that point (and all points listed in the `member_points` file) and report which component the point appears to lie on (up to the accuracy specified in `FINALTOL`).

A new development (as of the December 16, 2008 release of Bertini) is that witness point superset generation may now be carried out in three different ways. The default is the standard cascade algorithm, as described in Chapter 5. The other options are to use dimension-by-dimension slicing (with each dimension handled independently) and regeneration. The configuration setting WITNESSGENTYPE toggles between these options – please see the appendix for details.

2.2.3 User-defined homotopies

Bertini generally reads in the target system from the `input` file and immediately homogenizes it (unless it is homogenized), differentiates it, squares it up if necessary, adds patch equations (see

§4.1), adds a start system, and adds some form of homotopy (depending upon whether it is a zero- or a positive-dimensional run). However, the user has the option of providing a user-defined (i.e., parameter) homotopy, in which case none of this is done (except for automatic differentiation). In other words, if the user chooses to specify their own homotopy, they are responsible for any homogenization, squaring, and patch equations that are needed (in addition to the homotopy, start system, and starting solutions). Please note that in this case, a square system is necessary while homogenization and the use of patch equations is optional. The important configuration keyword to do this is `USERHOMOTOPY`.

In this case, the user must specify the variables using the keyword `variable` (since there are no groups involved as far as Bertini is concerned). It is also necessary to specify the name of the path variable (i.e., the variable through which the homotopy moves) and at least one parameter. The path variable may not appear in the definition of the functions, so if no parameter is actually desired, it is necessary to declare one parameter and set it equal to the path variable. Here is a simple `input` file to solve x^2 by moving from $x^2 - 1$ via a linear homotopy:

```
CONFIG
USERHOMOTOPY: 1;
END;
INPUT
variable x;
function f;
pathvariable t;
parameter s;
constant gamma;
gamma = 0.8 - 1.2*I;
s=t;
f = (x^2 - 1)*gamma*s + (x^2)*(1-s); %Note that only s appears, not t.
END;
```

There are several things to note in this `input` file that have not come up previously. First are the new keywords `parameter` and `pathvariable`. These keywords should only be used in this very special case of a user-defined homotopy in a zero-dimensional run. Also, this is the first time a comment has appeared in an `input` file in this manual. Note that the % sign is the comment character for Bertini; it causes Bertini to ignore the rest of that line (until a carriage return). White space (tabs and spaces) is ignored in Bertini. Finally, recall that it is not sufficient to provide only an `input` file for a user-defined homotopy. The user must also provide the list of starting points in a file called `start`. Here is the `start` file for this example:

```
2


-1.0 0.0;
```

```
1.0 0.0;
```

The first line indicates that there are two paths to be followed. Then the starting points are provided on separate lines, with the real part of each coordinate followed by a space, the imaginary part, and finally a semicolon. Points are separated by a single blank line.

If the two starting points for some system are $(2 + I, 1 - 3 \cdot I, -5)$ and $(1, 3.5 + I, 2 - 0.5 \cdot I)$ in three variables, the corresponding `start` file is as follows:

```
2

2.0 1.0;
1.0 -3.0;
-5.0 0.0;

1.0 0.0;
3.5 1.0;
2.0 -0.5;
```

Chapter 3

Syntax for input files

Syntax errors might be one of the most annoying parts of using any software! Care has been taken to provide some syntax-checking in Bertini, although the error message produced upon encountering a syntax error may be more general than hoped for by the user. Fortunately, the only input file with any sophisticated structure or rules is the `input` file, which is described in the next section. §3.2 might be useful if a syntax error message is reported. Finally, §3.3 covers the simple structure of the `start` and `member_points` files. Output files are described in Chapters 4 and 5.

3.1 The `input` file

As described in §2.2.1, the `input` file has two parts, grouped as follows (where the % symbol is the comment character in the `input` file, as usual):

```
CONFIG
% Lists of configuration settings (optional)
END;
INPUT
% Symbol declarations
% Optional assignments (parameters, constants, etc.)
% Function definitions
END;
```

The upper portion of the file consists of a list of configuration settings. Any configuration that is not listed in the `input` file will be set to its default value. A table of all configuration settings that may be changed, along with their default settings and acceptable ranges, may be found in Appendix A.

The syntax for the configuration lines is straightforward. It consists of the name of the setting (in all caps), followed by a colon, a space, the setting, and a semicolon. For example, to change the tracking type to 1 (the default is 0), simply include the following line in the `CONFIG` portion of the `input` file:

```
TRACKTYPE: 1;
```

The lower portion of the `input` file begins with a list of symbol declarations (for the variables, functions, constants, and so on). All such declarations have the same format:

```
KEYWORD a1, a2, a3;
```

where `KEYWORD` depends upon the type of declaration. All symbols used in the input file must be declared, with the exception of subfunctions. Here are details regarding each type of symbol that may be used in the input file:

- **FUNCTIONS**:

  Regardless of the type of run, all functions must be named, and the names must be declared using the keyword `function`. Also, the functions must be defined in the same order that they were declared.

- **VARIABLES**

  In all cases except user-defined homotopies, the variables are listed by group with one group per line, with each line beginning with either the keyword `variable_group` (for complex variable groups against which the polynomials have not been homogenized) or the keyword `hom_variable_group` (for variable groups against which the polynomials have been homogenized). Note that the user must choose one type of variable group for the entire input file, i.e., mixing of variable groups is not allowed in this release of Bertini. Also, only one variable group may be used for a positive-dimesional run. For example, if there are two nonhomogenized variable groups, the appropriate syntax would be

  ```
  variable_group z1, z2;
  variable_group z3;
  ```

  In the case of user-defined homotopies, the keyword is `variable`, and all variables should be defined in the same line.

- **PATHVARIABLES**:

  The pathvariable, often denoted by the letter "t", is the independent variable that is controlled during homotopy continuation. In Bertini, the homotopy always moves from the start system at $t = 1$ to the target system at $t = 0$. A pathvariable must be declared in the `input` file **ONLY** if the user is specifying the entire homotopy (i.e., `USERHOMOTOPY` is set to 1). In that case, it is also necessary to declare at least one parameter, as described in the next item. The keyword for pathvariables is `pathvariable`.

- **PARAMETERS**:

  Homotopy continuation relies on the ability to cast a given polynomial system as a member of a parametrized family of polynomial systems. Such parametrized families (especially those which occur naturally) constitute one of the powerful advantages numerical methods in algebraic geometry have over symbolic methods. Sometimes there is only one parameter involved, but sometimes there are several. Please note, though, that user-defined parameters

should be used only in the case of user-defined homotopies. Regardless of the number of parameters, each parameter depends directly upon the pathvariable. As a result, the user must both declare each parameter and assign to it an expression depending only upon the pathvariable to it. Here is an example:

```
...
parameter p1, p2;
...
p1 = t^2;
p2 = t^3;
....
```

For technical reasons, in the case of a user-provided homotopy, Bertini always assumes that there is at least one parameter (even if there is no apparent need for one). In the case that the user wishes to build a homotopy depending only upon the pathvariable, it is necessary to declare a parameter, set it to the pathvariable in the assignments section, and then to use only that parameter (and NOT the pathvariable) in the functions. Here is an example:

```
...
pathvariable t;
parameter s;
...
s=t;
....
```

No parameters should appear in the `input` file unless the homotopy is defined by the user, and the pathvariable should never appear in any homotopy.

- **CONSTANTS**:

  Bertini will accept numbers in either standard notation (e.g., 3.14159 or 0.0023) or scientific notation (e.g., 3.14159e1 or 2.3e-3). No decimal point is needed in the case of an integer. To define complex numbers, simply use the reserved symbol `I` for $\sqrt{-1}$, e.g., `1.35 + 0.98*I`. Please note that the multiplication symbol `*` is always necessary, i.e. concatenation does not mean anything to Bertini.

  Since it is sometimes useful to have constants gathered in one location (rather than scattered throughout the functions), Bertini has a `constant` type. If a constant type is to be used, it must be both declared and assigned to. Here is an example:

```
...
constant g1, g2;
...
g1 = 1.25;
g2 = 0.75 - 1.13*I;
....
```

Bertini will read in all provided digits and will make use of as many as possible in computations, depending on the working precision level. If the working precision level exceeds the number of digits provided for a particular number, all further digits are assumed to be 0 (i.e., the input is always assumed to be exact). This seems to be the natural, accepted implementation choice, but it could cause difficulty if the user truncates coefficients without realizing the impact of this action on the corresponding algebraic set.

- **SUBFUNCTIONS**:

  Redundant subexpressions are common in polynomial systems coming from applications. For example, the subexpression `x^2 + 1.0` may appear in each of ten polynomials. One of Bertini's advantages is that it allows for the use of subfunctions. To use a subfunction, simply choose a symbol, assign the expression to the symbol, and then use it in the functions. There is no need to declare subfunctions (and no way to do so anyway).

```
...
V = x^2 + 1.0;
...
f1 = 2*V^2 + 4.0;
....
```

3.2 Common syntax errors in the `input` file

Common complaints about Bertini are that (a) the parser that reads in the input is very picky and (b) the error messages are often to general. The development team agrees and will continue to work on this (especially during an upcoming complete rewrite). In the meantime, here is a list of syntax rules that are commonly broken, resulting in syntax errors:

- All lines (except `CONFIG` and `INPUT`, if used) must end with a semicolon.
- Bertini is case-sensitive.
- The symbol for $\sqrt{-1}$ is `I`, not `i`.
- In scientific notation, the base is represented by `e`, not `E`, e.g., `2.2e-4` is valid while `2.2E-4` is meaningless.
- For multiplication, * is necessary (concatenation is not good enough).
- Exponentiation is designated by `^`, not **.
- All symbols (except subfunctions) must be declared.
- No symbol can be declared twice. This error often occurs when copying and pasting in the creation of the `input` file.
- A pathvariable and at least one parameter are needed for user-defined homotopies. Please refer to the previous section for details.
- So that you don't worry, white space (tabs, spaces, and new lines) is ignored.

3.3 The `start` file

The structure of this file is very simple. In fact, it was described fully in §2.2.3 above, so please refer to that section for the basic syntax and structure. Although white space is ignored in the `input` file, it is important not to add extra spaces and/or lines to the `start` file. The `member_points` file uses the same syntax as the `start` file.

Chapter 4

Details about zero-dimensional runs

4.1 How Bertini finds these solutions

4.1.1 The basic idea

Bertini employs homotopy continuation to compute isolated solutions to polynomial systems. The idea of this technique is to cast the target polynomial system in a parametrized family of systems, one of which (the start system) has known or easily found solutions. After choosing this family, one chooses a path from the start system to the target system, constructs a homtopy between the two, and tracks the solution paths using predictor/corrector methods (e.g., Euler's method and Newton's method), employing adaptive steplength. This is a simplification of the true situation, of course. For complete details, please refer to the references of §1.1.

As described above, Bertini will automatically m-homogenize the target system if necessary and always performs automatic differentiation (except in the case of a user-defined homotopy). Bertini also squares the system up to the number of variables in the case of zero-dimensional runs, taking into account the m random patch equations that must be added to the system so as to work on patches in each projective space. Once all of this is done (if necessary), Bertini creates an m-homogeneous start system, solves it to obtain the start points (again, except in the case of a user-defined homotopy), and attaches the start and target systems into a homotopy (using the "gamma trick," which makes the appearance of singularities away from $t = 0$ a probability zero occurrence). At that point, Bertini carries out path-tracking on each path independently and reports the findings to the user, as described in the next two sections.

Please be aware that some paths could fail during tracking, possibly resulting in an incomplete set of solutions. For example, using fixed 128-bit precision but asking for 200 digits of accuracy will, of course, result in path failure. Not all causes of path failure are so clear, but there is often a set of tolerances that will allow convergence of all paths. The use of adaptive precision (MPTYPE: 2 plus some other settings – please see the appendix) will often help with convergence. Bertini reports to the screen the number of path failures, and there is a file named `failed_paths` describing each of the failures.

4.1.2 Endpoint sharpening for zero-dimensional runs

The sharpening module was designed to sharpen (refine) endpoints and display them to any number of digits. The sharpening module can be utilized either during a tracking run or afterwards. Since zero-dimensional runs can not currently determine if a singular solution is isolated, the sharpening module for zero-dimensional runs can only sharpen non-singular solutions. When using fixed precision, endpoints can only be sharpened to the number of digits of accuracy based on the precision. Using adaptive precision allows Bertini to automatically adjust the precision so that endpoints can be successfully sharpened to any desired number of digits. After sharpening, the endpoint is displayed using the smallest precision that is large enough to handle the number of digits requested.

When the sharpening module is utilized, all endpoints that fail to sharpen to the requested number of digits will be marked with a '&' in `main_data`.

4.1.3 Sharpen during tracking

During zero-dimensional tracking, the sharpening module can be used to sharpen and display the non-singular endpoints to any number of digits. That is, the paths are tracked based on the tracking tolerances and then all the non-singular endpoints are sharpened to the requested number of digits. To utilize the sharpening module during tracking, simply set SHARPENDIGITS to the desired number of digits.

4.1.4 Sharpen after tracking

After zero-dimensionl tracking has completed, the sharpening module uses the `raw_data` file to reprocess the endpoints and sharpen the non-singular ones to any number of digits. When the user utilizes sharpening in this way, a menu will be displayed that allows the user to either sharpen every endpoint, sharpen only the endpoints whose corresponding path numbers are listed in a file or allows the user to manually enter the path numbers of the endpoints to sharpen. When using a file of path numbers, the path numbers can only be separated with white space. If a non-number is reached, Bertini immediately stops reading in path numbers and attempts to sharpen the ones that were already read. For advanced users that use redirected input, please note that the input commands need to be put onto separate lines since buffer clearing is used for scanf stability.

To utilize the sharpening module after tracking, simply set SHARPENONLY to 1 and follow the menu.

4.1.5 Deflation

Deflation has been implemented in Bertini for the positive-dimensional case and will soon be available for the isolated singular endpoints found using a zero-dimensional run as well. Currently, the only way to deflated isolated singular endpoints is to use a positive-dimensional run to find the isolated singular endpoints.

4.1.6 Regeneration

Regeneration is an equation-by-equation method currently implemented for finding the non-singular isolated solutions. For more information on regeneration and equation-by-equation methods, see [16, 27].

To utilize regeneration, simply set USEREGENERATION to 1.

One feature of the equation-by-equation methods is that previous results affect future results. That is, if an error is made at a stage, it will influence the final results. Because of this, Bertini allows for the regeneration to be restarted from a previous run at any stage that was previously completed. To accomplish this, simply set REGENSTARTLEVEL to the place where you want to restart the regeneration from a previous run.

Please see the appendix for other regeneration specific settings.

4.2 Output on the screen

Bertini will output to the screen one or more tables classifying the solutions that it found. In particular, it will provide the number of real solutions, the number of nonsingular solutions, and so on. In the case that the user provides a nonhomogenized system, Bertini will also classify the points as finite or infinite and provide the previously-described data twice - once for the finite solutions and once for the infinite solutions. Since there is no obvious notion of infinity in the case of a homogenized target system, Bertini will just treat all points as finite in this case. The number of path crossings (if there were any) will also be reported to the screen. There is also a table of the number of solutions of each multiplicity. Perhaps the most useful information reported to the screen are a list of output files of interest and the number of paths that failed during the run.

4.3 Output in files

Depending on the case, Bertini will provide a number of files. The `main_data` file is of most interest to most users. It contains information about the solutions in a format that is easily read by humans. The file named `raw_data` contains similar data but is better suited for manipulation by software. In particular, the first two lines of `raw_data` (in order) are the number of variables and the number `0` (indicating the dimension), and the last line is `-1` (to indicate the end of the file). The interesting data lies between these lines and is broken into blocks with one block for each point. Each block has the following structure:

solution number[1]

bits of precision for the endpoint

the coordinates of the solution[2]

the norm of the function value at the endpoint

[1]The "solution number" does not necessarily correspond to the order of the start points in the case of a user-defined homotopy. The solutions are sorted after the run, and the resulting order is the source of this "solution number."

[2]Each coordinate gets its own line, with the real part listed before the imaginary part.

an estimate of the condition number at the most recent step of the path

the most recent Newton residual

the most recent value of the path variable, t

the error between the most recent estimates at $t = 0$

the value of the path variable when precision first increased

the cycle number used for the most recent estimate

success indicator (1 means successful, otherwise indicates a convergence problem)

A number of other (specialized) files are produced in some cases. The contents of these files should come as no surprise to the user given their names: `real_solutions`, `finite_solutions`, `nonsingular_solutions`, and so on. The structure of those files is much simpler than that of `raw_data`. In particular, the first two lines and the last line are the same, but the blocks for the points consist of only the solution number and the coordinate block. The file named `raw_solutions` has this structure but contains data for every solution.

Chapter 5

Details about positive-dimensional runs

5.1 How Bertini finds these solutions

Bertini carries out many of the same operations in this case as it does in the case of zero-dimensional tracking, as described in §4.1, so users should read the previous chapter before proceeding. Bertini (by default) also sets up the cascade homotopy (see the references of §1.1) and uses a total degree start system at the top level to get the cascade going[1]. Before all of this, though, Bertini computes the rank of the target system (via the SVD) at a generic point so that the system can be squared to the appropriate size (potentially decreasing both the number of variables and the number of polynomials), and so that the cascade will only run through dimensions that might have solutions.

The cascade algorithm produces a pure-dimensional witness superset in each dimension of interest, meaning it does not immediately find the desired witness sets, but rather sets that may contain "junk" from higher-dimensional components. This first step of positive-dimensional solving involves slicing the algebraic set of interest by generic hyperplane sections (in the form of linear equations with randomly chosen coefficients). As mentioned in §2.2.2, as of late December 2008, users may also ask Bertini to handle each dimension independently (i.e., not in a cascade) or use regeneration, via the configuration setting WITNESSGENTYPE. After witness superset generation is complete (running from codimension 1 to the codimension determined by the rank test), the "junk" is removed from each pure-dimensional witness point superset either by using a membership test or by using the local dimension test [3]. The configuration JUNKREMOVALTEST allows the user to select between the two junk removal algorithms. Once the junk is removed, Bertini has a witness point set for each dimension of interest. Using the witness set, the singular witness points are deflated. That is, for each witness point in the witness set, Bertini creates a system of polynomials and a witness point on a reduced component of the new system that corresponds to the given witness point.

To break the pure-dimensional witness sets into components, Bertini employs a combination of monodromy and a test known as the linear trace test. Details may be found in [28]. Once

[1]Bertini, as of December 2008, can also generate witness supersets via a dimension-by-dimension algorithm or regeneration. Please refer to WITNESSGENTYPE in the appendix for details.

the breakup is complete, all data is reported to the user and also stored in a special file named `witness_data`. Component sampling and membership testing then just involve reading in this file and moving the linears around either to sample a given component or to test whether the points given in `member_points` lie on an irreducible component of the algebraic set.

5.2 Output on the screen

There are two main parts of the output to the screen provided by Bertini in this case. One is a table giving the number of components, the total degree, the number of unclassified singular solutions, the number of paths, and the number of failed paths in each dimension of the cascade. The other is a table of components listed by dimension and by degree. As with the zero-dimensional case, Bertini will report an error message to the screen in the case of a suspected path crossing. Bertini detects this by comparing path values at $t = 0.1$ (or some other endgame breakpoint, as defined by the user). This is not a foolproof way of detecting path crossing, but it will catch many cases. There is also a list of files of interest.

5.3 Output in files

The main output file of interest is called `main_data`. It contains all points in the witness set in a format that is easily readable by humans. `witness_data` is a file suited for automated reading by a program. It contains all of the information needed to describe the witness set, in particular, the information used for component sampling and membership testing.

5.4 Component sampling and membership testing

For component sampling, Bertini reads in information about the witness set corresponding to the target system in the `input` file from the `witness_data` file. It then queries the user about which component to sample and how many points to find. After the user's choices are made, Bertini simply moves the appropriate linears around (as in monodromy) and gathers the desired number of sample points. These points are then simply returned to the screen.

For membership, Bertini reads in information about the witness set as in the case of sampling. It then reads in from the file `member_points` all points that the user would like to test. Bertini moves the appropriate slices around and reports to the screen whether any of the points lie on the algebraic set (and, if so, on which component they reside).

Chapter 6

Comments about running Bertini in parallel

Starting with the December 16, 2008 release of Bertini 1.1, parallel versions of Bertini were released. The parallel versions of Bertini were written using MPI and compiled using the MPICH and MPICH2 implementations of MPI 1.1 and MPI 2.0, respectively. For more details on MPICH and MPICH2, see http://www-unix.mcs.anl.gov/mpi.

The general syntax for running Bertini in parallel is

```
mpirun -np NUM_PROCESSES ./bertini FILE_NAME
```

where `NUM_PROCESSES` is the number of processes to use and `FILE_NAME` is the name of the input file. If `NUM_PROCESSES` is 1, Bertini runs in serial mode. Otherwise, Bertini uses 1 process as the head (master) process and the other processes as workers.

Chapter 7

A bit about how Bertini works

7.1 Data types and linear algebra

Of the available computational mathematics software packages (e.g., Maple and Matlab) and libraries (e.g., Linpack and LAPack) of which we are aware, only Maple supports multiprecision floating point arithmetic. However, to maximize control over various other aspects of the software, it was decided to write Bertini without relying on such existing software platforms. As a result, it was necessary to build basic data types and linear algebra subroutines, which are the topic of this section.

7.1.1 Basic data types

The GMP-based library called MPFR provides data types for arbitrary precision floating point numbers as well as basic declaration, assignment, and arithmetic functions. The basic floating point type in MPFR is the mpf_t data type, which contains a field for the precision of the instance and a several fields for the data itself. Using MPFR, bits may only be assigned in packets of 32.

The use of MPFR data types and functions is very computationally expensive, so it was necessary to have available in Bertini duplicate data types and functions. One set of types and functions uses entirely basic C double data types while the other set of types and functions uses the MPFR types and functions. Although this duality makes code-writing and maintenance somewhat tedious, it is a simple solution to the need for both types.

The most basic nontrivial data type in Bertini is the complex number, which is a structure consisting of two of the basic floating point type (double or mpf_t), named "r" and "i" for obvious reasons. Vectors (and the identical data type, points) and arrays are then built as structures consisting of an array of complex numbers and one or two integer fields containing the dimension(s) of the instance. Many more advanced structures are included in Bertini.

Adaptive precision introduces a special problem obviously not present with fixed precision runs: Bertini must somehow convert from regular precision types and functions to multiple precision versions. The solution to this problem used in Bertini is to have both versions of any given function available and to build a control function sitting over these two functions. When called, the control function will begin by calling the regular precision version of the function. The regular precision

version must be able to recognize when higher precision is needed using, for example, the criteria developed in [4, 5].

Once the regular precision version of the function at hand runs out of precision, it returns a special error code to the control function. The control function then uses special MPFR functions for copying regular precision data into MPFR data types to convert all necessary data from regular precision to the minimum level (64 bits) of multiple precision. After the conversion of all data to multiple precision types, the control function calls the multiple precision version of the function. That version of the function may then use the criteria to judge whether higher precision is needed, and if so, it can handle the precision increase itself, without reporting back to the control function.

There is at least one major drawback to this way of implementing the functions of numerical algebraic geometry. Every function comes in several versions. For example, for the power series endgame, there is currently a fixed regular precision version, a fixed multiple precision version, a version for regular precision that knows how to watch for precision problems, an analogous version for multiple precision, and a control function sitting over the adaptive precision versions. Whenever a bug is found or an extension is made, the developer must be very careful to correct or extend every version of the function.

It should also be noted that when one extends precision in MPFR, the added digits are chosen randomly. Although this may cause concern in some contexts, Bertini makes use of a refining function based on Newton's method to make certain the extended precision versions of all data types have adequate accuracy immediately following precision increases. Note that this is not a concern with the numerical data in the *input* file. Indeed, integers in *input* are stored exactly, and all floating point numbers are stored as rational numbers with denominators the appropriate factor of ten. Then, when straight-line program evaluation begins, these numbers are computed to the current working precision. The use of rational numbers to represent numerical data forces any digits added beyond the original precision of the input data to be set to zero, working under the assumption that the user is providing exact (or at least as exact as possible) information.

7.1.2 Linear algebra

Although more sophisticated techniques could be employed (e.g., Strassen-like matrix multiplication formulas), Bertini includes naive implementations of arithmetic and norms for matrices and vectors. The infinity norm is used almost exclusively throughout Bertini. Gaussian elimination with pivoting is used for matrix inversion, and failure is declared in that subroutine if any pivot is smaller than some threshold that has been hardcoded in the source code as a formula based on precision. As with the basic data types and most other sorts of subroutines, there are two copies of each linear algebraic operation, one in regular precision and one in arbitrary precision.

One of the endgames, the power series endgame, makes use of interpolation. Basic Hermite interpolation has been implemented since derivatives are readily available all along the path. These implementations of interpolation allow for an arbitrary number of data points.

The most sophisticated linear algebra routine in Bertini is the computation of the SVD of a matrix. The singular value decomposition of a matrix yields as a byproduct the condition number of the matrix, which is just the ratio of the largest and the smallest nonzero (interpreted numerically, of course) singular values. An implementation of the numerical computation of the singular value decomposition of a matrix is included in Bertini. [31] is another excellent resource regarding the

singular value decomposition.

7.2 Preprocessing of input files in Bertini

Before any actual tracking is carried out, Bertini passes through a preprocessing stage. It is during this part of the run that Bertini read in all of the data contained within the various input files and converts the polynomials of the input file to straight-line format. All such information is currently written to several files at the end of the preprocessing stage. The main part of Bertini, the tracker, then reads in these files at the beginning of the tracking phase.

The fundamental storage structure for polynomials in Bertini is the straight-line program. A straight-line program of a polynomial (or a function in general) is a list of unary and binary operations that may be used to evaluate the polynomial (or function). For example, the expression $(x+y)^2 * z$ may be broken into the following straight-line program:

$$t_1 = x + y\,,$$
$$t_2 = t_1^2\,,$$
$$t_3 = t_2 * z\,.$$

Then, to evaluate the expression, one may simply substitute in the values of x, y, and z as they are needed. The next section describes several advantages to using straght-line programs. The section after that describes specifically how Bertini converts polynomials in `input` into straight-line programs.

7.2.1 Advantages of using straight-line programs

There are several advantages to using straight-line programs for the representation of polynomial systems. For one, it allows for the use of subfunctions, as described above. Also, it sidesteps the difficulty of expression swell by allowing for non-standard expression representation, e.g., many more operations are needed to evaluate the expansion of $(x+y)^4$ (at least 17 operations) than are needed when evaluating the factored form of the expression. Finally, the use of straight-line programs makes automatic differentiation and homogenization very easy.

To homogenize a polynomial system given by a straight-line program, one must first record the breakup of the variables into m variable groups and add one new variable to each of the m groups. Then, when parsing the system into a straight-line program (as described in the next section), for each operation, one must simply compare the multidegrees of the operands and multiply in copies of the appropriate new homogenizing variables as needed. For example, if the operands of an addition have multidegrees $[0,1,0]$ and $[1,2,0]$, before recording the straight-line instruction for this operation, one must record two extra instructions corresponding to multiplying the first operand by the first and second homogenizing variables. After a system is homogenized, it is again just a matter of bookkeeping to produce the corresponding start system and to solve this start system in order to produce the set of start points.

Automatic differentiation is also very simple in the straight-line program setting. Please note that automatic differentiation is not the same as numerical or approximate differentiation -

the result of automatic differentiation is as exact as the input (even if the input is exact). The idea of automatic differenation is to produce a set of instructions for evaluating the derivative of a polynomial (or other function) given a straight-line program for the evaluation of the original polynomial (or function). There are many variations on the basic method, but the most basic method (called forward automatic differentiation) is used within a pair of loops to produce instructions for evaluating the Jacobian of a polynomial system in Bertini. For a good reference about automatic differentiation, please refer to [13].

After producing the straight-line program for a polynomial system, one may read through the set of instructions, writing down appropriate derivative evaluation instructions for each evaluation instruction. For example, given an instruction $x * y$ where x and y are memory locations, one could produce a set of three instructions for evaluating the derivative of this instruction. In particular, the first instruction would be to multiply the derivative of x (which should already have a memory location set aside) by y, the next would multiply x by the derivative of y, and the third would add these two, storing the result in the appropriate memory location. Naturally, there is a great deal of bookkeeping involved, but that is just a matter of careful implementation.

7.2.2 Implementation details for straight-line programs

Before Bertini performs path-tracking techniques or any other methods, it calls a parser to convert the Maple-style, human-readable input file into straight-line program. This preprocessing stage also includes automatic m-homogenization and automatic differentiation, as described in the previous section. Details of how to use lex and yacc (or the variants flex and bison) to parse polynomial systems into straight-line programs and details about the actual straight-line program data structure in Bertini are given in this section. To learn about the use of lex and yacc in general, please refer to [9].

To use lex and yacc (or their variants), one must produce two files that, together, describe how to convert the input structure into an output structure. For example, these files convert from Maple-style input files to straight-line programs (in the form of a list of integers) in Bertini. The main part of the lex file contains descriptions of the tokens to be detected within the input file. In the case of Bertini, these include tokens such as integers, floating point numbers, string names, and other implementation-specific words such as "variable_group" and "END".

The main part of the yacc file consists of an ordered set of rules to which the tokens may be set. There are a variety of nuances in creating such a yacc file, but the main idea is to create a list of unambiguous rules that together completely specify the structure of the input file. For example, the first two rules of the Bertini yacc file specify that the input file should have the structure of a "command" followed by a "command_list" or followed by the word "END." Each command is then a declaration or an assignment, and so on. The main part of this set of rules for Bertini is the definition of an "expression", the right hand side of a polynomial definition. Each expression is a sum of terms, each term is a product of monomials, each monomial is some kind of primary, and each primary is a number, a name, or another expression contained in a pair of parentheses.

Bertini does not only convert from Maple-style input to straight-line programs. Consider the example of $f = (x + y)^2 * z$ given above. By moving through the yacc rules, Bertini would first recognize $x + y$. Since x and y are variables that were presumably already declared before this statement, they have already been assigned addresses in an "evaluation array" of the complex

numbers type. A few constants like 0, 1, and $I = \sqrt{-1}$ are so important that they are given the first few addresses in this array, after which all other names and numbers that appear in the input file are given addresses. Assuming f, x, y, and z were already declared before this assignment statement, the evaluation array would look like:

index	0	1	2	3	4	5	6
name	i	0	1	f	x	y	z

The first instruction is then $t_1 = x + y$, written "+ t_1 x y" for convenience. This is then converted into address notation as "+ 7 4 5", i.e., the sum of the values in memory locations 4 and 5 is stored in memory location 7. When the 2 is encountered, it is also assigned an address in the evaluation array. In the end, the array is as follows, where IR means "intermediate result":

index	0	1	2	3	4	5	6	7	8	9	10
name	i	0	1	f	x	y	z	IR	2	IR	IR

To save work, all operations are assigned integer codes (in particular, their ASCII codes), so, for example, "+ 7 4 5" becomes "43 7 4 5." Table 6.1 lists all currently supported operations and their corresponding codes. Other operations, such as trigonometric functions of constants, are very simple to implement and will be included very soon.

Table 7.1: LEGAL OPERATIONS IN BERTINI

Operation	ASCII Code
+	43
- (binary)	45
*	42
/ (second operand must be numeric)	47
^ (second operand only must be integer)	94
=	61
- (unary)	78 (for N)

Thus, the Bertini-style straight-line program for the current example would be "43 7 4 5 94 9 7 8 42 10 9 6 61 3 10." If m-homogenization was being carried out during parsing, Bertini would simply compare the degrees of each operand while parsing and add extra instructions corresponding to multiplication by a homogenizing variable, as necessary. Once the straight-line program for a polynomial system has been created, a function called diff reads through the straight-line program one instruction at a time, creating instructions for the Jacobian of the system, as described above. Finally, Bertini concatenates the two straight-line programs to make one complete straight-line program for computing the function and derivative values of the polynomial system.

Bibliography

[1] E. Allgower and K. Georg. *Introduction to numerical continuation methods*, volume 45 of *Classics in Applied Mathematics*. Society for Industial and Applied Mathematics (SIAM), Philadelphia, PA, 2003. Reprint of the 1990 Spring-Verlag edition.

[2] D. Bates, I. Fotiou, and P. Rostalski. A numerical algebraic geometry approach to nonlinear constrained optimal control. To appear in *2007 IEEE Conf. on Decision and Control Proc.*, 2007.

[3] D. Bates, J. Hauenstein, C. Peterson, and A. Sommese. A local dimension test for numerically approximated points on algebraic sets. Preprint, 2008.

[4] D. Bates, J. Hauenstein, A. Sommese, and C. Wampler. Adaptive multiprecision path tracking. *SIAM J. Numer. Anal.*, 46(2):722–746, 2008.

[5] D. Bates, J. Hauenstein, A. Sommese, and C. Wampler. Stepsize control for adaptive multiprecision path tracking. Preprint, 2008.

[6] D. Bates, C. Peterson, and A. Sommese. A numerical-symbolic algorithm for computing the multiplicity of a component of an algebraic set. *J. Complexity*, 22:475–489, 2006.

[7] D. Bates, C. Peterson, and A. Sommese. Applications of a numerical version of Terracini's Lemma for secants and joins. In IMA Volume 146: Algorithms in Algebraic Geometry (eds. A. Dickenstein, F. Schreyer, and A. Sommese), pages 133-152, 2008.

[8] D. Bates, C. Peterson, A. Sommese, and C. Wampler. Numerical computation of the genus of an irreducible curve within an algebraic set. Preprint, 2008.

[9] D. Brown, J. Levine, and T. Mason. *lex & yacc (A Nutshell Handbook)*. O'Reilly Media, Sebastopol, CA, 1992.

[10] B. Dayton, Z. Zeng. Computing the multiplicity structure in solving polynomial systems. In *ISSAC 2005 Proc.*, pages 116–123, 2005.

[11] T. Gao and T.Y. Li. HOM4PS. Software available at www.csulb.edu/∼tgao.

[12] T. Gao, T.Y. Li, and M. Wu. Algorithm 846: A software package for mixed-volume computation. *ACM Trans. Math. Software*, 31(4):555-560, 2005. Software avialable at www.csulb.edu/∼tgao.

[13] A. Griewank. *Evaluating derivatives: principles and techniques of algorithmic differentiation*, volume 19 of *Frontiers in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2000.

[14] A. Griewank and M. Osborne. Analysis of Newton's method at irregular singularities. *SIAM J. Numer. Anal.*, 20(4):747–773, 1983.

[15] T. Gunji, S. Kim, M. Kojima, A. Takeda, K. Fujisawa, and T. Mizutani. PHoM—a polyhedral homotopy continuation method for polynomial systems. *Computing* 73(1):55–77, 2004. Software available at www.is.titech.ac.jp/∼kojima.

[16] J. Hauenstein, A. Sommese, and C. Wampler. Regeneration of polynomial systems. Preprint, 2008.

[17] A. Leykin, J. Verschelde, and A. Zhao. Evaluation of jacobian matrices for newton's method with deflation for isolated singularities of polynomial systems. In *SNC 2005 Proc.*, pages 19–28, 2005.

[18] T.-Y. Li. Numerical solution of multivariate polynomial systems by homotopy continuation methods. In *Acta numerica, 1997*, volume 6 of *Acta Numer.*, pages 399–436. Cambridge Univ. Press, Cambridge, 1997.

[19] Y. Lu, D. Bates, A. Sommese, and C. Wampler. Finding all real points of a complex curve. To appear in Contemp. Math., 2008.

[20] A. Morgan. *Solving polynomial systems using continuation for engineering and scientific problems*. Prentice-Hall Inc., Englewood Cliffs, NJ, 1987.

[21] A. Morgan, A. Sommese, and C. Wampler. Computing singular solutions to nonlinear analytic systems. *Numer. Math.*, 58(7):669–684, 1991.

[22] A. Morgan, A. Sommese, and C. Wampler. Computing singular solutions to polynomial systems. *Adv. in Appl. Math.*, 13(3):305–327, 1992.

[23] A. Morgan, A. Sommese, and C. Wampler. A power series method for computing singular solutions to nonlinear analytic solutions. *Numer. Math.*, 63(3):391–409, 1992.

[24] A. Sommese, J. Verschelde, and C. Wampler. Numerical decomposition of the solution sets of polynomial systems into irreducible components. *SIAM J. Numer. Anal.*, 38(6):2022–2046, 2001.

[25] A. Sommese, J. Verschelde, and C. Wampler. Homotopies for intersecting solution components of polynomial systems. *SIAM J. Numer. Anal.*, 42(4):1552–1571, 2004.

[26] A. Sommese, J. Verschelde, and C. Wampler. Introduction to numerical algebraic geometry. In *Algorithms and Computation in Mathematics*, Volume 14 (eds. A. Dickenstien and I.Z. Emiris), pages 339–392, 2005.

[27] A. Sommese, J. Verschelde, and C. Wampler. Solving polynomial systems equation by equation. In IMA Volume 146: Algorithms in Algebraic Geometry (eds. A. Dickenstein, F. Schreyer, and A. Sommese), pages 133-152, 2008.

[28] A. Sommese and C. Wampler. The numerical solution of polynomial systems arising in engineering and science. World Scientific Publishing Co. Pte. Ltd., Hackensack, NJ, 2005.

[29] G. Stewart. *Matrix algorithms. Vol. I: Basic decompositions*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1998.

[30] H.-J. Su, J. McCarthy, M. Sosonkina, and L. Watson. Algorithm 857: POLSYS_GLP – A parallel general linear product homotopy code for solving polynomial systems of equations. *ACM Trans. Math. Software* 32(4):561–579, 2006. Software available at www.vrac.iastate.edu/∼haijunsu.

[31] L. Trefethen and D. Bau. *Numerical linear algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[32] T. Turocy. Towards a black-box solver for finite games: finding all Nash equilibria with Gambit and PHCpack. To appear in IMA Volume on Software for Algebraic Geometry (eds. M. Stillman, N. Takayama, and J. Verschelde), 2008.

[33] J. Verschelde. Algorithm 795: PHCpack: a general-purpose solver for polynomial systems by homotopy continuation. *ACM Trans. Math. Software* 25(2):251–276, 1999. Software available at www.math.uic.edu/∼jan.

[34] J. Verschelde and Y. Wang. Numerical homotopy algorithms for satellite trajectory control by pole placement. In *MTNS 2002 Proc.*, CD, 2002.

[35] J. Verschelde and Y. Wang. Computing dynamic output feedback laws. *IEEE Trans. Automatic Control* 49(8):1393–1397, 2004.

[36] C. Wampler. Numerical algebraic geometry and kinematics. In *SNC 2007 Proc.*, 2007.

[37] C. Wampler. HomLab: Homotopy continuation lab. Software available at www.nd.edu/∼cwample1.

[38] J. Wilkinson. *Rounding errors in algebraic processes*. Dover Publications Inc., New York, 1994. Reprint of the 1963 Prentice-Hall original.

[39] Z. Zeng. ApaTools – A software toolbox for approximate polynomial algebra. Software available at www.neiu.edu/∼zzeng/apatools.htm.

Appendix A

Configurations

A.1 Optional configurations

There are presently roughly 50 configurations that the user can change in Bertini. That is beneficial in that more options yield the ability to attack more problems. It is a drawback, though, in that 50 configurations is a large number to understand and master. Fortunately, the typical user will care about no more than 20 of them. As a result, those settings of most importance are given in Table A.1 while those that are also available but may not be as interesting to the typical user are given in Table A.2. Each setting is also described briefly in this section (in the same order that they appear in the table).

- TRACKTYPE

  Set to 0 for zero-dimensional tracking, 1 for numerical irreducible decomposition, 2 for component sampling, or 3 for the component membership test.

- MPTYPE

  Indicates the level of precision to be used. Set to 0 for regular precision, 1 for higher precision, or 2 for adaptive precision. If MPTYPE=1, the user should also set PRECISION to the desired level of precision. Note that in the case of MPTYPE=2, Bertini will automatically begin each path with regular precision, regardless of the setting of PRECISION. Also, if MPTYPE=2, COEFFBOUND, DEGREEBOUND, AMPSAFETYDIGITS1, and AMPSAFETYDIGITS2 should be set up; please refer to the descriptions of those settings for details.

- PRECISION

  This indicates the level of precision to use (in bits) if MPTYPE=1. Standard settings include 64 bits (roughly 19 decimal digits), 96 (28), 128 (38), 160 (48), 192 (57), 224 (67), and 256 (77). In general, N bits is equivalent to $\lfloor N \frac{\ln(2)}{\ln(10)} \rfloor$ decimal digits. The maximum is 3328 bits (roughly 1000 digits), and this setting has no effect if MPTYPE=0 or 2.

- TRACKTOLBEFOREEG

Table A.1: Configuration settings of particular interest

NAME	ACCEPTABLE VALUES	DEFAULT VALUE
TRACKTYPE	0, 1, 2, 3	0
MPTYPE	0, 1, 2	0
PRECISION	$\geq$64 and $\leq$3328	96
TRACKTOLBEFOREEG	$>$0	1e-5
TRACKTOLDURINGEG	$>$0	1e-6
FINALTOL	$>$0	1e-11
MAXNORM	$>$0	1e5
MINSTEPSIZEBEFOREEG	$>$0 and $\leq$MAXSTEPSIZE	1e-14 - fixed precision
MINSTEPSIZEDURINGEG	$>$0 and $\leq$MAXSTEPSIZE	1e-15 - fixed precision
IMAGTHRESHOLD	$>$0	1e-8
COEFFBOUND	$>$0	1000
DEGREEBOUND	$>$0	5
TARGETTOLMULTIPLIER	$\geq$1	10
AMPMAXPREC	$\geq$64 and $\leq$3328	1024
CONDNUMTHRESHOLD	$>$0	1e8
PRINTPATHMODULUS	$\geq$0	20
RANDOMSEED	$\geq$0	0
SINGVALZEROTOL	$>$0	1e-12
USERHOMOTOPY	0, 1	0
SHARPENDIGITS	$\geq$0	0
SHARPENONLY	0,1	0
WITNESSGENTYPE	0,1,2	0
JUNKREMOVALTEST	0,1	0
REDUCEDONLY	0,1	0
ENDGAMENUM	1,2	1
NUMSAMPLEPOINTS	$\geq$2 and $\leq$32	2
USEREGENERATION	0,1	0
SECURITYLEVEL	0,1	0
SECURITYMAXNORM	$>$0	1e4

Table A.2: Other configuration settings

NAME	ACCEPTABLE VALUES	DEFAULT VALUE
SCREENOUT	0, 1	0
OUTPUTLEVEL	-1, 0, 1, 2, 3	0
STEPSFORINCREASE	$\geq$1	5
MAXNEWTONITS	$\geq$0	2
MAXSTEPSIZE	$>$0 and $\leq$1	0.1
NBHDRADIUS	$>$0 and $<$1	1e-100
ENDGAMEBDRY	$>$0 and $<$1	0.1
TARGETTIME	$\geq$0 and $<$1	0
MAXNUMBERSTEPS	$>$0	10000
SAMPLEFACTOR	$>$0 and $<$1	0.5
MAXCYCLENUM	$\geq$1	6
AMPSAFETYDIGITS1	anything	1
AMPSAFETYDIGITS2	anything	1
MAXNUMPTSFORTRACE	$>$0	5
MAXNUMMONLINEARS	$>$0	5
MAXNUMBADLOOPSINMON	$>$0	10
INTRINSICMULTIPLIER	$\geq$0 and $\leq$1	0.75
REGENSTARTLEVEL	$\geq$0	0
REGENREMOVEINF	0,1	1
SLICETOLBEFOREEG	$>$0	1e-7
SLICETOLDURINGEG	$>$0	1e-8
SLICEFINALTOL	$>$0	1e-11

The desired tracking tolerance during the non-endgame portion of the path. In other words, the goal of the corrector steps is to have two iterates within TRACKTOLBEFOREEG of each other (using the infinity norm).

- TRACKTOLDURINGEG

  The same as TRACKTOLBEFOREEG, except that it is used in the endgame portion of the path.

- FINALTOL

  This is the tolerance to which the endpoints of successful paths will be accurate. For example, if FINALTOL is $1e-11$, the first 11 digits of the endpoint will be correct.

- MAXNORM

  For post-processing, this is the cutoff for finite versus infinite endpoints. Also, for user-defined homotopies, if any point on the path exceeds MAXNORM, tracking will be terminated since the path appears to be going to infinity.

- MINSTEPSIZEBEFOREEG

  This is the smallest steplength that Bertini will allow during the non-endgame portion of each path. If a predictor/corrector step fails and causes the steplength to drop below this level, the path will be declared a failure. Since adaptive precision verifies the precision is sufficient for the step size, the user should not adjust this configuration when using adaptive precision.

- MINSTEPSIZEDURINGEG

  The same as MINSTEPSIZEBEFOREEG, except that it holds in the endgame portion of the path.

- IMAGTHRESHOLD

  Endpoints are considered real if the imaginary parts of the coordinates are all less than IMAGTHRESHOLD in absolute value.

- COEFFBOUND

  When using adaptive precision (MPTYPE=2), it is necessary to provide a bound on the sums of the absolute values of the coefficients of the polynomials. In other words, COEFFBOUND must bound the sum of the absolute values of the coefficients of every polynomial in the polynomial system.

- DEGREEBOUND

  As with COEFFBOUND, DEGREEBOUND must be provided whenever adaptive precision (MPTYPE=2) is used. DEGREEBOUND is simply a bound on the degrees of the polynomials in the system. In the very near future, this setting will be automatically set based on the polynomial system.

- TARGETTOLMULTIPLIER

  Two endpoints will be considered equal if they differ by less than the final tolerance (FINALTOL) times a factor, named TARGETTOLMULTIPLIER.

- AMPMAXPREC

  This is the maximum precision (in bits) allowed when using adaptive precision (to avoid infinit loops.

- CONDNUMTHRESHOLD

  For basic zero-dimensional tracking, this is the cutoff for determining singular versus non-singular. For regeneration and positive-dimensional tracking, singular versus non-singular is based on the corank of the Jacobian rather than a fixed bound.

- PRINTPATHMODULUS

  This is the frequency with which Bertini will print out path numbers while running. It has no effect on the run. The idea is that infreqeunt updates (e.g., PRINTPATHMODULUS=0, i.e., the path number is never printed to the screen) leaves the user wondering how the run is progressing, while frequent updates can be tedious. Only path numbers divisible by PRINTPATHMODULUS will be printed (unless it is set to 0).

- RANDOMSEED

  If set to zero, different random numbers will be chosen for each run. Otherwise, the random number generator will be seeded with RANDOMSEED, allowing the user to repeat runs identically.

- SINGVALZEROTOL

  A singular value will be considered zero if it is less than SINGVALZEROTOL in absolute value. This effects the rank determination at the beginning of each zero- and positive-dimensional run.

- USERHOMOTOPY

  Set to 0 to use automatic homotopies, 1 to use user-defined homotopies.

- SHARPENDIGITS

  When using the zero-dimensional solver, setting this to a positive integer will instruct Bertini to sharpen the non-singular solutions to that many (decimal) digits. When using component sampling, setting this to a positive integer will instruct Bertini to sharpen the sample points to that many (decimal) digits.

- SHARPENONLY

  Set to 0 to use regular tracking methods, 1 to start the sharpening module. The sharpening module uses either the witness data from a positive-dimensional solve or the raw data from a zero-dimensional solve to sharpen solutions to any number of digits.

- WITNESSGENTYPE

  Set to 0 to use the standard cascade algorithm for witness superset generation, 1 to have Bertini handle each dimension separately, and 2 to have Bertini use regeneration.

- JUNKREMOVALTEST

  Set to 0 to use a membership test for junk removal and set to 1 to use the local dimension test of [3].

- REDUCEDONLY

  Set to 1 to find only reduced components during a positive-dimensional run, 0 to find all components.

- ENDGAMENUM

  Set to 1 to use the fractional power series endgame and set to 2 to use the Cauchy endgame.

- NUMSAMPLEPOINTS

  The number of sample points used in the endgame to create the approximation. With 2 sample points, the power series endgame generates a 3rd order approximation.

- USEREGENERATION

  Set to 1 to use regeneration in a zero-dimensional run, 0 otherwise. During a positive-dimensional run, setting to 1 uses regeneration to compute a witness superset.

- SECURITYLEVEL

  Set to 0 to have Bertini truncate paths that appear to be headed to infinity, 1 otherwise.

- SECURITYMAXNORM

  When SECURITYLEVEL is 0, this is the cutoff to truncate paths. That is, if two successive approximations are larger than SECURITYMAXNORM, the path is truncated.

- SCREENOUT

  Set to 1 to have all output printed to the screen as well as to the appropriate output file. Otherwise set to 0. A setting of 0 should suffice unless debugging.

- OUTPUTLEVEL

  The levels -1 to 3 provide a different amount of path tracking information to the `output` file. A setting of 0 should suffice unless debugging.

- STEPSFORINCREASE

  This is the number of consecutive successful predictor/corrector steps taken before attempting to increase the steplength.

- MAXNEWTONITS

  This is the maximum number of corrector steps allowed after any given predictor step. Experience indicates that this should be set no higher than 3 and with 2 being more effective at avoiding pathcrossing.

- MAXSTEPSIZE

  This is the largest steplength that Bertini will allow during the run as well as the starting steplength.

- NBHDRADIUS

  Tracking will halt if the value of the pathvariable comes within NBHDRADIUS of the target value of the pathvariable.

- ENDGAMEBDRY

  This is the value of the pathvariable at which the tracking method being employed switches from basic path-tracking to the indicated endgame.

- TARGETTIME

  This is the desired value of the pathvariable, almost always 0.

- MAXNUMBERSTEPS

  This is the most steps that may be taken during the endgame on any one path (to avoid wasting computational power on paths that are not converging well).

- SAMPLEFACTOR

  This is the factor used to determine the geometrically-spaced sample points for all of the endgames. For example, if ENDGAMEBDRY=0.1 and SAMPLEFACTOR=2, the sample points will be 0.1, 0.05, 0.025, etc.

- MAXCYCLENUM

  In the power series endgame, this provides a basic idea of what cycle numbers to test.

- AMPSAFETYDIGITS1

  The first of the two safety digit settings for adaptive precision. Bertini will go to higher precision the higher that these are set. Conversely setting these to be negative is not recommended, but will likely make Bertini track the paths faster.

- AMPSAFETYDIGITS2

  See AMPSAFETYDIGITS1.

- MAXNUMPTSFORTRACE

  Monodromy will be used to break pure-dimensional witness sets into irreducible components until either the maximum number of monodromy loops (see the next two settings) is exceeded or the number of unclassified points drops below MAXNUMPTSFORTRACE. Either way, the exhaustive trace test will then be called to finish the classification combinatorially.

- MAXNUMMONLINEARS

  This is the maximum number of different linears to try during monodromy before giving up and going to the exhaustive trace test.

- MAXNUMBADLOOPSINMON

  This is the stopping criterion for using each monodromy linear. In particular, new loops will continue to be chosen using the same target linear until either all points are classified

(or at least enough are - see MAXNUMPTSFORTRACE) or there have been MAXNUMBADLOOPSINMON consecutive useless loops. A loop is useless if it does not yield any new groupings among the points.

- INTRINSICMULTIPLIER

  Regeneration and dimension-by-dimension slicing for numerical irreducible decomposition can use intrinsic slicing. This setting determines when to switch from intrinsic slicing to extrinsic slicing. For example, a setting of 0.75 instructs Bertini to use intrinsic slicing when the ratio of variables using intrinsic versus extrinsic formulations is less than 0.75. In particular, 0 means totally extrinsic and 1 means totally intrinsic.

- REGENSTARTLEVEL

  If you are restarting a regeneration run, use this setting to start at a previously completed regeneration level.

- REGENREMOVEINF

  Set to 1 to remove infinite endpoints, 0 otherwise.

- SLICETOLBEFOREEG

  Similar to TRACKTOLBEFOREEG, but for moving slices during regeneration.

- SLICETOLDURINGEG

  Similar to TRACKTOLDURINGEG, but for moving slices during regeneration.

- SLICEFINALTOL

  Similar to FINALTOL, but for moving slices during regeneration.