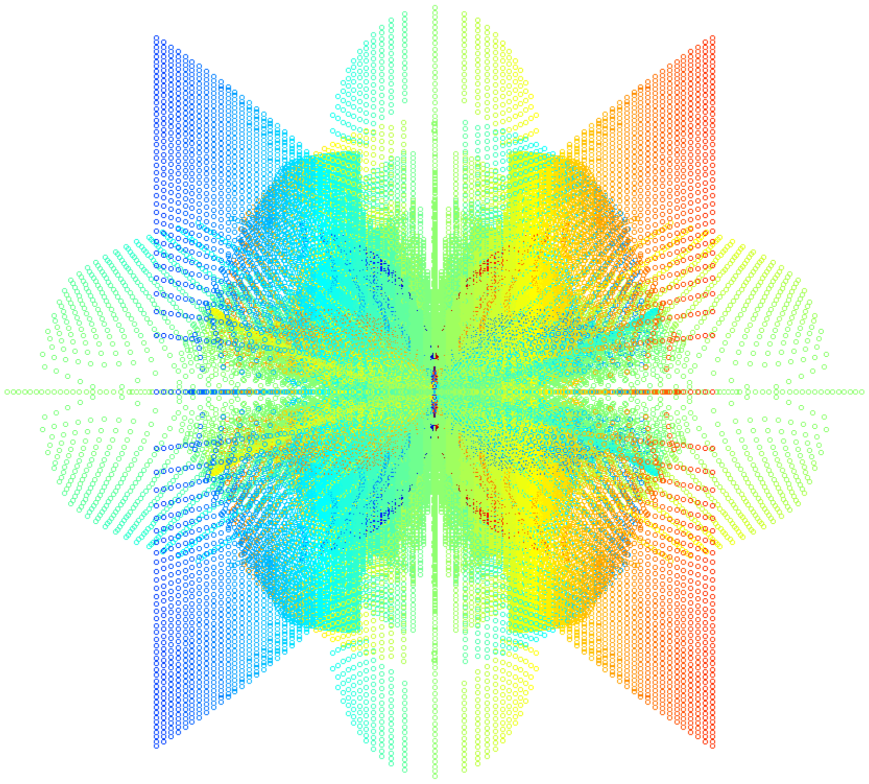
Paramotopy

Parallel Parameter Homotopy
via Bertini



Manual by
Daniel Brake
Colorado State University
Mathematics
Fort Collins, Colorado

Contents

1 Introduction

The Paramotopy program is a compiled linux/unix wrapper around Bertini which permits rapid parallel solving of parameterized polynomial systems. It consists of two executable programs: 'Paramotopy', and 'Step2', and further depends on having a copy of the parallel version of Bertini. Paramotopy is called from the command line, and in turn calls Bertini and Step2.

Briefly, homotopy continuation is the tracking of solutions from one system to another through continuous deformation; a simplified example appears in Figure 1. Using a combination of prediction and correction, Bertini performs such a deformation. First, it obtains the solutions to the initial system. Taking discrete steps in complex-valued time, each solution path is tracked individually. If suitable options are chosen in the configuration of the run, precision will be automatically adjusted to compensate for loss of accuracy near system singularities due to inversion of poorly conditioned matrices. Additionally, Bertini has various "endgame" schemes for bringing the paths to successful completion, as well as determining whether paths crossed or other problems happened along the way. Nearly all these options may be accessed through Paramotopy. For a thorough description of the Bertini program, see the Bertini User's Manual.

Paramotopy is implemented to make use of both the basic zero-dimensional solve Bertini performs by default, as well as exploiting the `userhomotopy` mode. An illustration of the scheme is in Figure 2. First, Paramotopy performs what we define here to be a 'Step1' run. Bertini finds all zero-dimensional solutions to the system supplied to it, for a set of parameter values randomly determined; for this solve, there will likely be superfluous paths. Second, Paramotopy tracks all meaningful solutions found in Step1 (from `nonsingular_solutions`), to each parameter point at which the user wishes to solve; this is called 'Step2'. Performing the second step in this way has the potential to eliminate the extra paths from Step1.

With respect to the soundness and reliability of this method, theory dictates that off a set of measure zero, an algebraic system will have the same number of (complex) roots throughout its parameter space. Therefore, we are virtually guaranteed that for *randomly* chosen point in parameter space, we will find the full generic number of solutions. Then for Step2, as we track from the random point to the specific points at which we wish to solve, we will find all the solutions.

Paramotopy uses a specially compiled library of Bertini, MPICH2 for process distribution, OpenMP for accurate timing measurements, TinyXML for preference and information storage, and Boost for filesystem operations. System requirements and compilation information may be found in Section 2. In Section 3 is presented the format for the input files. For information on configuring and using Paramotopy, see Section 3. Coarse troubleshooting tips are in Section 6.

Contact

For assistance with Paramotopy, please contact Daniel Brake at `danielthebrake@gmail.com`.
For help with Bertini, contact Daniel Brake `danielthebrake@gmail.com`, Dan Bates `bates@math.colostate.edu`, or one of the other Bertini authors.

Acknowledgements

This research utilized the CSU ISTeC Cray HPC System supported by NSF Grant CNS-0923386.

disclaimer

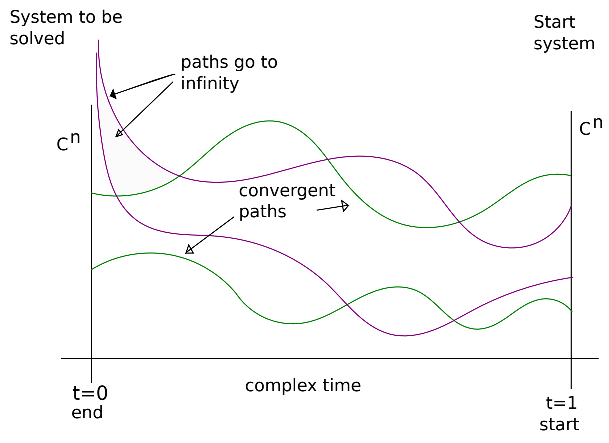


Figure 1: Generic Homotopy Continuation

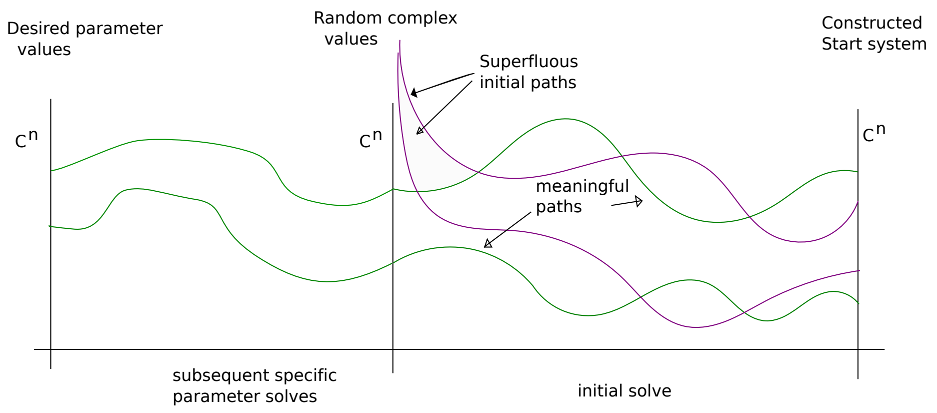


Figure 2: Parameter Homotopy, with initial 'Step1' run.

2 Getting Started

2.1 Compilation and Installation

Paramotopy has the following library dependencies:

- Tinyxml (lies internally to the program's folders, as per the license agreement).
- MPI, version2 (for parallelism)
- Boost – regex, system, filesystem (for file manipulation)
- OpenMP (for the most accurate timing available).

In addition, Paramotopy relies on a suitably compiled version of Bertini, which has the following dependencies:

- mpfr
- gmp
- Bison
- Flex
- MPI (version2)

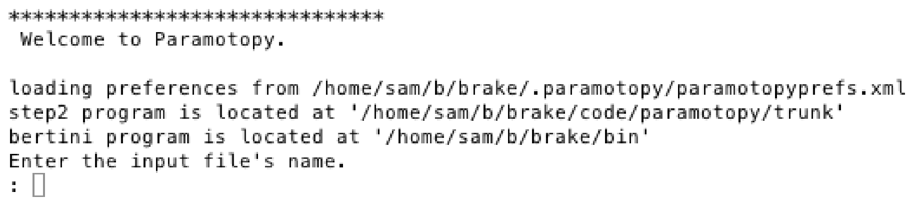
When Bertini is compiled for use with Paramotopy, it must be compiled in two ways: into an executable, and a library. When compiled into the executable, it must have the flag `-D_HAVE_MPI` (for use with Paramotopy). In contrast, the library *must not* have the `-D_HAVE_MPI` flag. In addition, we redefine `main()` to be `bertini_main` via the command `-Dmain=bertini_main`. Compilation is a little different of every machine, of course, and a future milestone is to use something like `./configure` to handle setting everything up properly for start-to-finish with minimal user input.

As it stands now, Paramotopy is compiled using a Makefile. Various options are available in the makefile:

- `MACHINE`: choose the name of the machine you are compiling on. The makefile contains a section for each machine, and sets the necessary variables suitably. To compile on a new machine, create a new section, with all the necessary options.
- `TIMING`: choose `YES` or `NO`. Conditional compilation using `#ifdef` in the code turns on or off timing statements for Step2.
- `OPT`: Sets available compiler options, such as the `-O` optimization flag, and compiler warning levels. These are left user-set, as every compiler has different options.
- `VERBOSE`: Chooses whether Step2 ought to be compiled with `#ifdefs` for extra output to the screen.

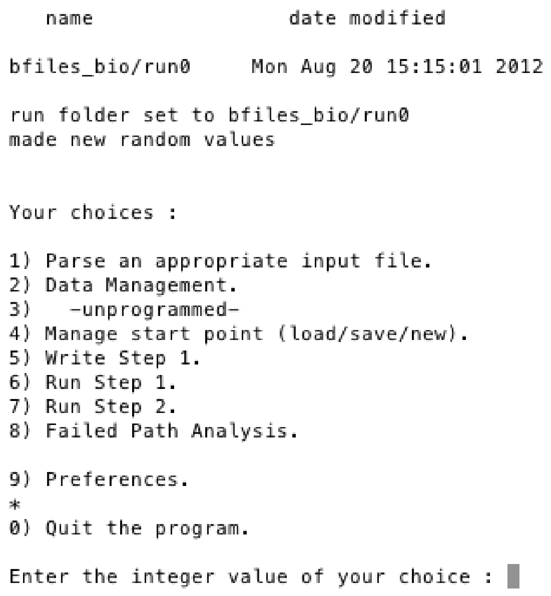
Simply issuing the `make` command should be enough to compile the programs once initial configuration is completed.

2.2 Using Paramotopy



```
******************************
 Welcome to Paramotopy.

loading preferences from /home/sam/b/brake/.paramotopy/paramotopyprefs.xml
step2 program is located at '/home/sam/b/brake/code/paramotopy/trunk'
bertini program is located at '/home/sam/b/brake/bin'
Enter the input file's name.
: 
```

Figure 3: The welcome screen. If you do not supply the filename as the first argument to Paramotopy from the command line, it will immediately prompt you for the name. You cannot use the program without an input file.



```
   name                date modified

bfiles_bio/run0     Mon Aug 20 15:15:01 2012

run folder set to bfiles_bio/run0
made new random values


Your choices :

1) Parse an appropriate input file.
2) Data Management.
3)   -unprogrammed-
4) Manage start point (load/save/new).
5) Write Step 1.
6) Run Step 1.
7) Run Step 2.
8) Failed Path Analysis.

9) Preferences.
*
0) Quit the program.

Enter the integer value of your choice : 
```

Figure 4: Main menu, describing the available choices. Use numeric input, or whatever it prompts for. If you just loaded the program, it will describe what directory it is working in.

3 Input Files

The input file format for Paramotopy is fairly simple. Illustrative examples appear in Files 1,2,3. It consists of:

1. Declare the numbers of: functions `a`, variable groups `b`, parameters `c`, and constants `d`.

   `a b c d`

2. Functions declaration, without any name declarations, or terminating semi-colons.
3. Variable groups come next, with each group getting its own line; variables are comma separated, and there is *no terminating character*.
4. Constant declaration, with the word `constant` appearing before a comma separated, semi-colon terminated line containing the constant names. Each constant gets its own line, with `name = value;` being the format.
5. Finally, the type of solve is indicated:
   - 0 indicates the computer will supply a mesh. The final lines tell the name of the parameter, starting point of discretization, ending point of discretization, and number of discretization points. Note that the format for a line of parameter declaration is

     `name a b c d e`,

     where startpoint = `a`+`b`i, endpoint = `c`+`d`i, and `e` indicates the number of discretization points. See File 2.
   - 1 asserts the user supplies a text file containing parameter value. If it is userdefined, the next line is the name of the file, and the final lines simply indicate the parameter names. See File 3.

```
nfunct nvargrp nconst nparam      <- all four must be present
function1
function2                         <- no terminating semicolon
...                               <- do not name functions
function n                        <- no '=' sign
vargroup1
vargroup2
constant c1,c2,...,ck;            <- note semicolon, omit if nconst=0
c1 = 0.1234;                      <- semicolon and equal sign
c2 = 0 + 9*I;                     <- capital 'I' for sqrt(-1)
...
ck = 1-1*I;
0                   or    1       <- choose mesh or userdefined
p1   0 0 1 0 10     or    p1      <- declare mesh discretization
p2   0 0 1 0 64     or    p2            or parameter name
...
pj   0 0 1 0 13     or    pj
```

File 1: Generic Paramotopy input file.

Paramotopy has minimal error correction in the portion of the program, so an error in an input file is likely to cause a fairly benign (will not corrupt data) program crash. If the program is able to parse the input file without errors, it will display information to the screen and user may check everything has been imported correctly. This will be corrected in a future version.

```
   7 1 2 21
   Fa-k1*a*b+k2*c-k4*a*f+k5*f-Da*a
 3 Fb-k1*a*b+k2*c+k8*g-k9*b*f-Db*b
   Fc+k2*a*b-k2*c-k3*c-k6*c*e+k7*g-Dc*c
   Fd+k3*c-Dd*d
   Fe-k4*a*e+k5*f-k6*c*e+k7*g-De*e
   Ff+k4*a*e-k5*f+k8*g-k9*b*f-Df*f
 8 Fg+k6*c*e-k7*g-k8*g+k9*b*f-Dg*g
   a,b,c,d,e,f,g
   constant k3,k1,k5,k6,k7,k8,k9,Fa,Fb,Fc,Fd,Fe,Ff,Fg,Da,Db,Dc,Dd,De,Df,
       Dg;
   k3=.80349;
   k1=.980389;
13 k5=.8034;
   k6=.8018;
   k7=.16876;
   k8=.7982;
   k9=.58973;
18 Fa=.4264;
   Fb=.5284;
   Fc=.1687;
   Fd=.167896;
   Fe=.5673;
23 Ff=.69386;
   Fg=.79827;
   Da=.0692;
   Db=.08762;
   Dc=.2897;
28 Dd=.0828;
   De=.26967;
   Df=.4238;
   Dg=.5872;
   0
33 k4    0  0  1  0  40
   k2    0  0  1  0  50
```

File 2: Paramotopy input file demonstrating use of user-defined parameter file. Line 1 indicates 12 equations (lines 2-13), in 1 variable group (line 14), with five parameters (lines 33-34), and 21 constant declarations (10-31). On line 32, `0` tells Paramotopy to make a mesh from the parameters discretized in lines 33-34. Parameter `k4` will be broken into 40 points on the complex line segment $0+0i$ to $1+0i$. Similarly, `k2` will be broken into 50 points, so a solve using this input file would have 2000 points total in the Step2 run.

```
 1 12 1 5 0
   0.4318*c1*c2 - 0.2435*s1 + 0.0934*a*s1 + 0.0203*a*c1*s2*s3 - 0.0203*a*
       c1*c2*c3 - (0.4318*c4*c5 - 0.1501*s4 - 0.0203*c4*c5*c6 + 0.4331*c4
       *c5*s6 + 0.4331*c4*c6*s5 + 0.0203*c4*s5*s6+delta)
   0.2435*c1 - 0.0934*a*c1 + 0.4318*c2*s1 + 0.0203*a*s1*s2*s3 - 0.0203*a*
       c2*c3*s1 - (0.1501*c4 + 0.4318*c5*s4 - 0.0203*c5*c6*s4 + 0.4331*c5
       *s4*s6 + 0.4331*c6*s4*s5 + 0.0203*s4*s5*s6)
   0.0203*a*c2*s3 - 0.4318*s2 + 0.0203*a*c3*s2 - (0.4331*c5*c6 - 0.4318*
       s5 + 0.0203*c5*s6 + 0.0203*c6*s5 - 0.4331*s5*s6)
   0.4318*c1*c2 - 0.1501*s1 - 0.0203*c1*c2*c3 + 0.4331*c1*c2*s3 + 0.4331*
       c1*c3*s2 + 0.0203*c1*s2*s3 - x
 6 0.1501*c1 + 0.4318*c2*s1 - 0.0203*c2*c3*s1 + 0.4331*c2*s1*s3 + 0.4331*
       c3*s1*s2 + 0.0203*s1*s2*s3 - y
   0.4331*c2*c3 - 0.4318*s2 + 0.0203*c2*s3 + 0.0203*c3*s2 - 0.4331*s2*s3
       - z
   s1^2+c1^2-1
   s2^2+c2^2-1
   s3^2+c3^2-1
11 s4^2+c4^2-1
   s5^2+c5^2-1
   s6^2+c6^2-1
   s1,c1,s2,c2,s3,c3,s4,c4,s5,c5,s6,c6
   1
16 robomc_10000
   a
   delta
   x
   y
21 z
```

File 3: Paramotopy input file demonstrating use of computer-generated mesh, and constant declaration. Line 1 indicates 7 equations (lines 2-8), in 1 variable group (line 9), with two parameters (lines 17-21), and zero constant declarations. Line 15's `1` indicates Paramotopy should look for the file on the next line, titled `robomc_10000`. The name of the file is arbitrary, but should be at the same path at the input file.

3.1 Monte Carlo Input

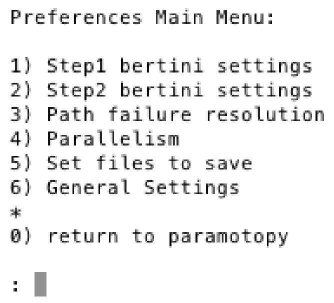
Paramotopy enables the user to supply their own plain text file of parameter points over which to solve a parametrized family of polynomials. See File 3 for an example of such an input file, and File 4 for an example of the user-defined file.

```
1 0  1.56133 0 −0.156326 0 −0.611479 0 0.430234 0
1 0  1.56133 0 −0.423045 0 −0.178196 0 0.498284 0
1 0  1.56133 0 −0.683401 0 0.0113754 0 0.35804 0
1 0  1.56133 0 0.560595 0 0.178767 0 0.0282532 0
1 0  1.56133 0 −0.391407 0 −0.00876144 0 0.464572 0
1 0  1.56133 0 0.218755 0 0.530148 0 −0.429091 0
1 0  1.56133 0 −0.277281 0 0.620464 0 −0.18738 0
1 0  1.56133 0 0.436947 0 −0.116395 0 −0.503699 0
1 0  1.56133 0 0.464191 0 −0.185408 0 −0.365677 0
1 0  1.56133 0 0.183529 0 0.348453 0 0.344025 0
1 0  1.56133 0 0.327672 0 0.047893 0 −0.608647 0
1 0  1.56133 0 −0.0981337 0 0.505425 0 0.0346073 0
1 0  1.56133 0 −0.595407 0 −0.204762 0 0.601097 0
1 0  1.56133 0 0.798536 0 0.1768 0 −0.193611 0
1 0  1.56133 0 0.501227 0 −0.465084 0 0.515747 0
1 0  1.56133 0 0.234762 0 0.466421 0 −0.370907 0
1 0  1.56133 0 −0.0867984 0 0.540995 0 0.518395 0
1 0  1.56133 0 −0.477477 0 0.125201 0 0.561676 0
```

File 4: User-defined parameter point file. This file is named in the input file (in relation to file 3, this file is `robomc_10000`), and must be placed in the same directory. Each line indicates one parameter point, and each real-complex pair is separated by a space.

4 Options & Configuration

Persistent configuration of Paramotopy is maintained through the `$HOME/.paramotopy/paramotopyprefs.xml` file located in the home directory. The following sections describe what the settings affect. Note that since the preferences file is merely text, the user could manually hack it with a text editor.

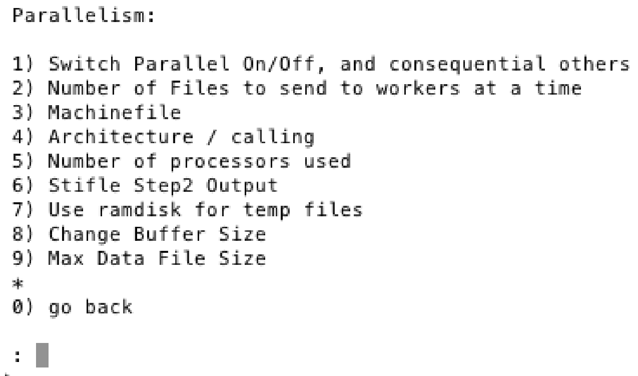


```
Preferences Main Menu:

1) Step1 bertini settings
2) Step2 bertini settings
3) Path failure resolution
4) Parallelism
5) Set files to save
6) General Settings
*
0) return to paramotopy

:
```

Figure 5: The Paramotopy main menu. Interaction is via mainly numeric input at the console.

4.1 Parallelism

Parallelism is achieved using MPICH2. Paramotopy itself acts as a gateway to the Step2 program, where all the real work is done. The gateway allows the user to load input files, make new data folders, change settings, run Bertini for Step, call Step2, and perform Failed Path Analysis on a completed Step2 run. The gateway model was used because Bertini uses MPI itself, and `MPI_Init()`, `MPI_Finalize()` can only be called once within a program. Therefore, Step2 uses the nonparallel version, whereas Step1 can call either one.



```
Parallelism:

1) Switch Parallel On/Off, and consequential others
2) Number of Files to send to workers at a time
3) Machinefile
4) Architecture / calling
5) Number of processors used
6) Stifle Step2 Output
7) Use ramdisk for temp files
8) Change Buffer Size
9) Max Data File Size
*
0) go back

:
```

Figure 6: Parallelism menu.

- `Switch Parallel On/Off`
  turn on or off parallel solve mode. If off, then a single processor will be used for all portions of the program
- `Number of Files to send to workers at a time`
  to minimize network traffic, it is advisable to distribute the work in chunks. Setting this value

close to one will kill performance, while setting it near the total number of parameter points will send all the work to a single processor. You must find balance.

- `Machinefile`
  some MPI installations require a machine file for process distribution.

- `Architecture / calling`
  set the call to start an MPI process on your machine. Two defaults are built in: `mpiexec`, `aprun`. You may also use your own, although there is no error checking for whether your custom command exists, or functions properly. On the todo list is adding custom calling sequences for more complex command strings.

- `Number of processors used`
  Set the number of processors for Step1, Step2, and rerunning of failed paths (realized as Step2 runs). There should be at least two processors for this mode, as Paramotopy currently uses the master-slave model, with a single master. If you wish to use only one process, switch off parallel mode.

- `Stifle Step2 Output`
  Bertini produces an overwhelming amount of screen output. Not only is this annoying, but sending text to the screen overwhelms the network. It is suggested to stifle only when you are confident in the success of your Step2 runs. Stifling by redirection to `/dev/null` produces a significant speedup.

- `Use ramdisk for temp files`
  Bertini uses hard disk IO to communicate with itself and other processes. This can overwhelm the hard disk, uses read/write cycles reducing the lifetime of a hard drive, and injures performance. Using a location in memory as a hard disk, otherwise known as a ramdisk, will grant you enormous performance gains. Two locations are scanned for existence by default: `/dev/shm` and `/tmp`. If either are found, the user is prompted for confirmation. The user may also choose a custom location. However, at present time the name of the temporary file location root directory must be the same across all computers used. Note: Paramotopy does not guarantee that the chosen location is usable in the desired fashion; the user must ensure that the temporary location is read/writable.

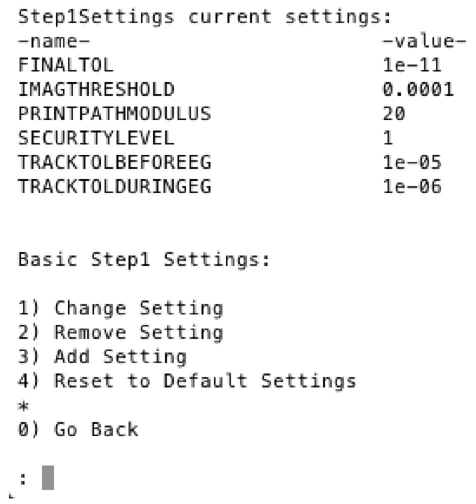
- `Change Buffer Size`
  to minimize writes to the hard drive, each worker buffers its files in memory, and writes to disk only when a threshold is reached. To change this threshold, change this buffer size.

- `Max Data File Size`
  The text files produced by Paramotopy grow in time, of course. To ensure that you don't produce unwieldly files, change the maximum file size for data files. The default is 64MB. This maximum is not a *hard* maximum, in the sense that a new file is started only after the max is achieved.

4.2 Step1 and Step2 Bertini Settings

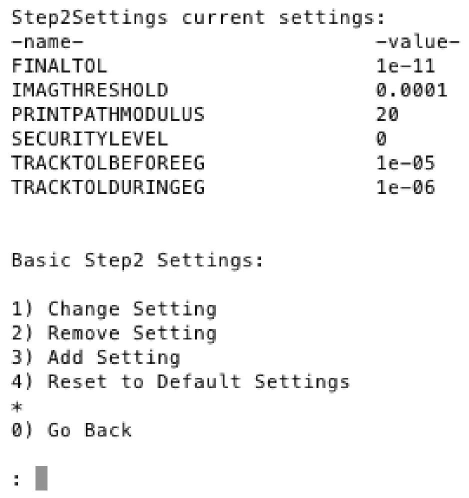
Separate settings categories are present for both the Step1 and Step2 runs, and persist from run to run, and session to session. These are written directly into the input files for Bertini, and there is no error checking – if the user sets a setting to one disallowed, or misspells a name, Step1 or Step2 will fail.

Options are to change, delete, and add a setting. User may also reset all of that Step's settings, and reset to default, which are the Bertini default values as well. Note that each setting has a type, being either a string, integer, or double value. Doubles may use the `1e-4` format for convenience.



```
Step1Settings current settings:
-name-                        -value-
FINALTOL                      1e-11
IMAGTHRESHOLD                 0.0001
PRINTPATHMODULUS              20
SECURITYLEVEL                 1
TRACKTOLBEFOREEG              1e-05
TRACKTOLDURINGEG              1e-06


Basic Step1 Settings:

1) Change Setting
2) Remove Setting
3) Add Setting
4) Reset to Default Settings
*
0) Go Back

:
```

Figure 7: Step1 Settings. All settings are direct Bertini config options.



```
Step2Settings current settings:
-name-                        -value-
FINALTOL                      1e-11
IMAGTHRESHOLD                 0.0001
PRINTPATHMODULUS              20
SECURITYLEVEL                 0
TRACKTOLBEFOREEG              1e-05
TRACKTOLDURINGEG              1e-06


Basic Step2 Settings:

1) Change Setting
2) Remove Setting
3) Add Setting
4) Reset to Default Settings
*
0) Go Back

:
```

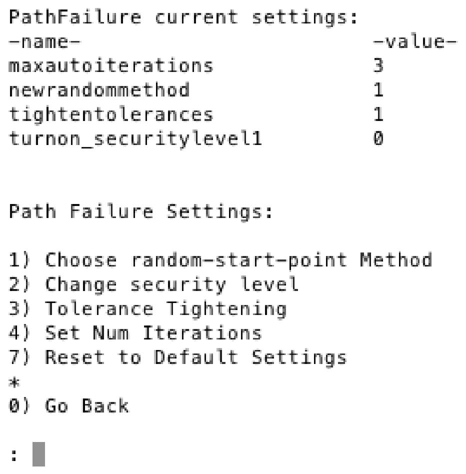
Figure 8: Step2 Settings. All settings are direct Bertini config options.

While capitalization does not matter in Bertini, Paramotopy is currently case-sensitive, so deleting or changing a setting requires the same capitalization as was used initially.

4.3 Path Failure

Paramotopy detects failed paths that occur during a run, and has methods for resolving the system, repeatedly if necessary, to get the proper solution set.

- `Choose random-start-point Method`
  When resolving points with failed paths, Paramotopy can optionally choose a new start point before solving the set of fails. To be developed is a variety of methods for choosing this start point; *e.g.* away some distance from the old, etc.



```
PathFailure current settings:
-name-                        -value-
maxautoiterations             3
newrandommethod               1
tightentolerances             1
turnon_securitylevel1         0


Path Failure Settings:

1) Choose random-start-point Method
2) Change security level
3) Tolerance Tightening
4) Set Num Iterations
7) Reset to Default Settings
*
0) Go Back

: 
```

Figure 9: Path Failure Settings menu. The path failure analysis section of the program utilizes the Bertini preferences from Step1 and Step2 sections, optionally changing them with successive runs.

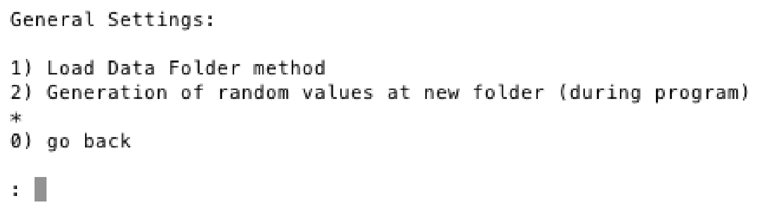
- `Change security level`
  ensure `securitylevel=1` is turned on, for more secure solution of failed paths.

- `Tolerance Tightening`
  When Failed Path Analysis is entered in Paramotopy, the settings are those currently used for Step2 runs. If tolerance tightening is enabled, at each iteration of resolve, the `finaltol`, `tracktolbeforeeg`, and `tracktolduringeg` values are decreased by a factor of ten.

- `Set Num Iterations`
  Path Failure mode will automatically try to resolve failed paths [up to] this number of times before returning back to the user for input.

- `Reset to Default Settings`
  Reset PathFailure settings to default hardcoded values.

4.4 General

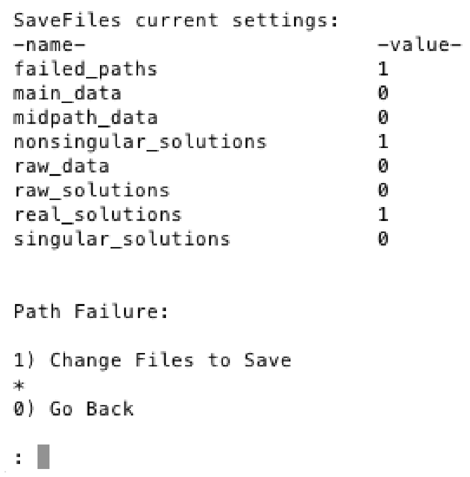


```
General Settings:

1) Load Data Folder method
2) Generation of random values at new folder (during program)
*
0) go back

: 
```

Figure 10: General Settings menu. Sets behaviour for such things as start point generation at creation of a new folder, and whether to create a new data folder at launch.

- `Load Data Folder Method`
  at launch of Paramotopy, if not supplied with an argument of the filename to load, Paramotopy will ask the user for the filename. Each filename gets its own folder for storing data, and in this folder will be another folder containing the data for a run. At launch, Paramotopy can do one of the following:
  - Reload the most recently used run's data,
  - Make a new run folder, and make new start point,
  - Ask the user what to do.
- `Generation of random values at new folder (during program)`
  If while running the program, the user wishes to start a new run, should Paramotopy:
  - Keep the random values currently in use, or
  - Make a new set of random values.

4.5 File Saving



```
SaveFiles current settings:
-name-                        -value-
failed_paths                  1
main_data                     0
midpath_data                  0
nonsingular_solutions         1
raw_data                      0
raw_solutions                 0
real_solutions                1
singular_solutions            0


Path Failure:

1) Change Files to Save
*
0) Go Back

:
```

Figure 11: Paramotopy can save each of the output files from Bertini, for each point at which it solves the input system. Failed paths is always saved; the rest are optional.

Bertini produces several output files which may be desirable to keep around. The file named `failed_paths` is always saved, to enable Failed Path Analysis. All other files are optional. Some are more useful than others for humans or computers. For a more thorough description of the output of Bertini, see the Bertini User's Manual.

5 Data

Paramotopy is capable of producing immense amounts of data. Probably the two most useful files you can choose to save are `nonsingular_solutions` and `real_solutions`. Both are easy to have a machine parse, with predictable numbers of lines per parameter point.

Some points in parameter space will be faster to solve than others. Combined with the use of a parallel machine, this will result in data files which are out of order. Whatever data collection method you use, you must deal with this one fact.

Data collection into text files is achieved by simply copying the Bertini output file into memory, and then, along with the index of the parameter point and the parameter values, into a collective output file. The format is essentially what appears in File 5:

```
   Pr1  Pi1  Pr2  Pi2
 2 0
   0  0  0  0  0  0  0  0
   2

   −0.2008901478456057752876258730604423765803e−40
        −0.8609577764811676083755394559733244710586e−41
 7 0.8609577764811676083755394559733244710586e−41
        0.2188267681889634337954496117265533030607e−40

   0.000000000000000e+00  7.589415207398531e−19
   7.589415207398531e−19  −9.351243737687476e−19

12 1
   0.555556  0  0  0  0  0  0  0
   2

   −7.318364664277155e−19  9.520650327138336e−19
17 1.218033378151684e−18  0.000000000000000e+00

   −0.8121868202266301032981141853923093549866e−18
        0.8704335440385715417597808916363135312587e−18
   0.1229193841385470029124013154420321923155e−17
        0.1420916030667928865394846908826236211630e−19
```

File 5: Example output file from Paramotopy. The first line of each file created by Paramotopy declares the names of the parameters. Each point gets its own section, with the index (lines 2, 12), parameter values (lines 3, 13), and copied Bertini output. This is an example of `real_solutions` output, so lines 4,14 indicate the number of solutions which follow. The actual solutions appear as well. Note that lines 4-10, and 14-20, are unmodified Bertini output.

The author Daniel Brake has some generic MATLAB code which can do basic parsing of some file types, as well as data display methods. If you need help with any aspect, don't hesistate to ask!

6 Troubleshooting

- Step1 fails
  - Are all the settings properly spelled, and allowed?
  - Was the calling sequence correct?
  - Do you have the *parallel* version of Bertini installed?
- Step2 fails
  - Are all settings spelled properly and allowed?
  - Was the calling sequence correct?
  - Did Paramotopy correctly parse your input file?
  - Did the temporary files write to an acceptable location?