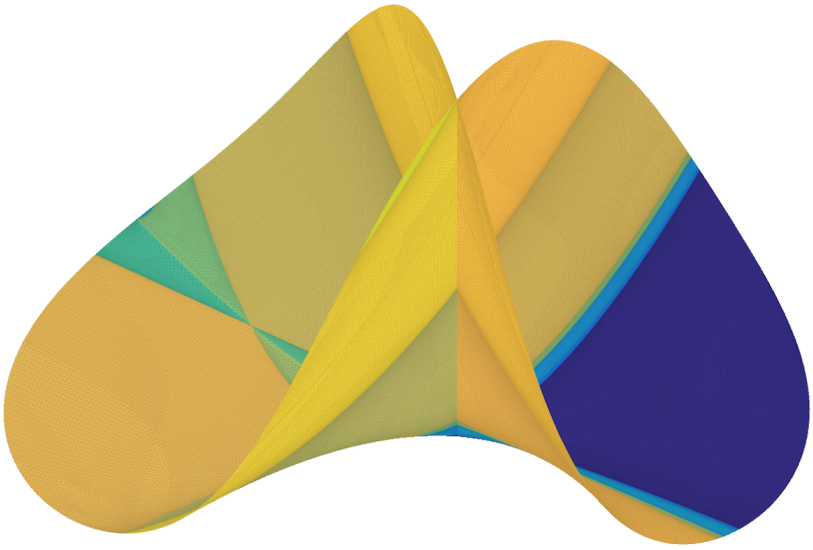
Bertini_real

Software for Real Algebraic Sets

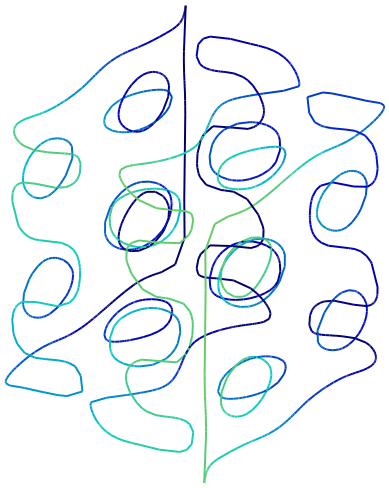
A Comprehensive Guide



This manual compiled May 6, 2019

Manual written by

Danielle Brake
Pierce Cunneen
Elizabeth Sudkamp
Christopher Lembo



Brought to you by the power of numerical algebraic geometry

Contents

1 Introduction

Welcome to Bertini_real, software for real algebraic geometry. This manual is intended to help the user operate this piece of numerical software, to obtain useful and high-quality results from decomposing real algebraic curves and surfaces.

Bertini_real is compiled software, links against a parallel version of Bertini 1 compiled as a library (`libbertini-parallel`, and requires Matlab and the Symbolic Computation toolbox. It also requires several other libraries, including a few from Boost, and an installation of MPI. All libraries should be compiled using the same compilers and dependent libraries.

1.1 About this manual

The purpose of this manual is to provide a robust, orderly, and easy to understand instructions on how to use Bertini_real. This manual has three roles: it first serves as a description of what Bertini_real is, followed by instructions on Bertini_real's installation on Mac, Linux, and PC operating systems, and finally as a general reference manual for Bertini_real.

This manual is here to help guide a user through the installation process, as well as act as the user's manual for Bertini_real. If there is a section that might not be entirely clear, or is confusing to a reader, please contact us (see below for contact information), and we will try to resolve the problem. Such feedback is welcome!

1.2 Bertini_real product description

Bertini_real is an implementation of several numerical algorithms [8, 3], to decompose the real part of a complex curve or surface in any (tractible) number of variables. Some of the important features of Bertini_real include :

- It is a command line program for numerically decomposing the real portion of a one- or two-dimensional complex irreducible algebraic set in any reasonable number of variables.
- It seeks to automate the visualization and computation of algebraic curves and surface.

1.3 Where Bertini_real can be found

The code for Bertini_real can be found at GitHub. Take a recursive clone to pick up submodules. You have to build it from source yourself; pre-compiled binaries are not offered.

1.4 Who is developing Bertini_real?

Bertini_real is under ongoing development by the development team, which consists of

- Danielle Brake (University of Wisconsin - Eau Claire, main implementer)
  danielleamethystbrake@gmail.com

- Dan Bates (US Naval Academy, Advisor)
- Jonathan Hauenstein (University of Notre Dame, Advisor)
- Wenrui Hao (Penn State, Non-self-conjugate case)
- Andrew Sommese (University of Notre Dame, Advisor)
- Charles Wampler (General Motors. R&D, Advisor)
- Pierce Cunneen (University of Notre Dame, Windows use)
- Nicolle Ho (Notre Dame, Python curve visualization)
- Elizabeth Sudkamp (Notre Dame, Windows use, symbolics in Python)

This manual was written by Danielle Brake, Pierce Cunneen, Chris Lembo, and Elizabeth Sudkamp.

1.5 Acknowledgements

The development of Bertini_real has been supported generously by a number of sources, including

- the Vincent J. and Annamarie Duncan Professor of Mathematics, at the University of Notre Dame,
- the University of Notre Dame,
- the National Science Foundation grants DMS-1025564 , DMS-1115668, and DMS-1262428,
- the Air Force Office of Scientific Research grant FA8650-13-1-7317,
- Mathematical Biosciences Institute,
- the Sloan Research Fellowship,
- the Army Young Investigators Project, and
- the Defense Advanced Research Projects Agency Young Faculty Award.

Finally, funding for shared facilities used in this research was provided by the Division of Computer and Network Systems: an NSF grant under award number CNS-0923386.

Disclaimer

2 Quick Summary

Here's a super brief description of how to use Bertini_real.

1. Open a terminal or command prompt. Navigate to the directory containing the `input` file.

2. Run Bertini on an input file using the `tracktype:1` setting. This is done by typing in the command line: `bertini` with an input file named `input`. Bertini will produce a Numerical Irreducible Decomposition that will be used by Bertini_real.

3. Run Bertini_real on the same input file. Similarly, just type `bertini_real` in the command line. Bertini_real will provide a cellular decomposition of the real portion of a one- or two-dimensional complex algebraic set.

4. Consider running the `sampler` to smooth your decomposition.

5. Visualize the results of Bertini_real in Matlab. Enter Matlab and call `gather_br_samples`, which parses the output results of into a .mat file, and then call `bertini_real_plotter`, which will plot the curve or surface in Matlab.

Notes

Please note that

- either the Matlab executable must be on the path, with the symbolic toolbox;

- or, you have Python installed with supporting modules as necessary. I don't remember what those modules are right now. `pip install` is your friend. Decompose Whitney and install dependencies til it works ;)

Bertini_real can be

- cloned from https://github.com/ofloveandhate/bertini_real. Clone recursively.

Use of Bertini_real depends on Bertini, which itself has several important dependencies (see Appendix B). Once installed, you can run Bertini_real from the command line.

3 Input Files

The instructions provided go through how to create input files, run these files through Bertini and Bertini_real, and view a graphical representation of the results using MATLAB. Most of this information about Bertini, its input files, and syntax will be taken or paraphrased from the slightly-out-of-date Bertini User's Manual, which can be read here.

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)
tracktype:1; % needed in order to run Bertini_real
END;
INPUT
% Symbol declarations
% Optional assignments (parameters, constants, etc.)
% Function definitions
END;
```

File 1: Adapted from [1]

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 the Appendix.

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;
```

File 2: Adapted from [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;
```

File 3: Adapted from [1]

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-dimensional run. For example, if there are two nonhomogenized variable groups, the appropriate syntax would be

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

File 4: Adapted from [1]

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

- 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:

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

File 5: Adapted from [1]

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;
...
```

File 6: Adapted from [1]

- SIN, COS, PI, AND EXP: Starting with Bertini v1.2, the sine function sin, cosine function

cos and exponential function exp are built into Bertini. Additionally, Bertini uses `Pi` for the constant π. To avoid confusion with scientific notation, the constant e is not specifically built in Bertini, but the user can define their own constant and set it equal to `exp(1)`, as shown below.

```
...
constant EN; % Euler's number e
EN = exp(1);
...
```

File 7: Adapted from [1]

It is important to note that Bertini will return an error if the argument of `sin`, `cos`, or `exp` depends upon a variable when trying to solve a polynomial system. There is no such restriction for user-provided homotopies.

3.1 On Bertini input syntax

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`. If you prefer to use `i`, you may define `i` as a subfunction by including the statement `i = I;`.
- In scientific notation, the base is represented by `e` or `E`, e.g., 2.2e-4.
- For multiplication, * is necessary (concatenation is not enough).
- Exponentiation is designated by ^.
- All symbols except subfunctions must be declared prior to use. You cannot combine declaration and definition, sadly.
- 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.
- White space (tabs, spaces, and new lines) is ignored.

4 Preliminary: running Bertini

4.1 Numerical Irreducible Decomposition

Bertini_real takes as input a Numerical Irredicible Decomposition (NID) of the complex algebraic variety for your problem. The NID is computed by Bertini, `tracktype: 1`, and is stored in a file called `witness_data`.

Once your input file describing the system you want to solve is created, you need to run Bertini. Navigate in the command line to the directory of the `input` file and type `bertini` or `bertini your_input_file_name`. Bertini assumes the input file is named `input` unless told otherwise, by passing the name as the first argument. No, you do not currently use flags to specify input file name with Bertini 1.

Cygwin users: A user may also use `bertini-serial.exe` (or `bertini_parallel.exe`). This will run Bertini, creating the Numerical Irreducible Decomposition needed for Bertini_real. You may need to type in the entire pathway to where Bertini is located, if it's not in the same folder, so the command line read
`/cygdrive/path/to/BertiniSource_v1.5/bertini-serial.exe input`

If the NID run is successful, you should see a summary of the decomposition print to the screen. It should look something like this:

```
************* Witness Set Decomposition *************

| dimension | components | classified | unclassified
-------------------------------------------------------
|   2       |   1        |   7        | 0
-------------------------------------------------------

************** Decomposition by Degree **************

Dimension 2: 1 classified component
-------------------------------------------------------
   degree 7: 1 component

*******************************************************
```

File 8: Example NID screen output, tracktype 1 in Bertini 1

If there are path failures or unclassified points, change Bertini settings, and re-run the problem. Consult the Bertini book or user's manual for more information about available settings, and their impact on computing the NID.

4.2 Necessary Bertini output files for Bertini_real

The main output file of interest from Bertini to feed into Bertini_real is called `witness_data`, a file suited for automated reading by a program. It's terribly formatted for humans. See the Bertini book [2] for information about what is contained in `witness_data`.

Shortly, `witness_data` contains all of the information needed to describe the witness sets for the irreducible components of your variety. In particular, it has the information used for regeneration used in Bertini_real, as well as component sampling and membership testing.

Do not rename `witness_data`. Bertini_real will do its best to preserve this file against loss. If your `witness_data` took a lot of effort to compute, you are encouraged to not use the original data file as input to Bertini_real, or any other program. Archive the original, and use a duplicate.

5 Running Bertini_real

Bertini_real is called from the command line. This is done simply by calling `bertini_real` (or `bertini_real.exe` for Cygwin users) from the command line. If the input file is called anything other than `input`, then the `-input` or `-i` option followed by the filename must be used.

It is important to note that for Bertini_real to run, the MATLAB executable must be on the path.

Bertini_real uses the tracker options for Bertini, which are set at the top of the input file, in the `CONFIG` section.

We suggest the following configuration options in the input file for Bertini_real:

- `sharpendigits` ≈ 30

  helps keep regeneration start points on target, and helps identify points which are supposed to be the same point.

Other options can improve performance and tighten up the produced decomposition.

If you do not have MATLAB installed you may run Bertini_real using python. To run Bertini_real using Python run the command `bertini_real symengine python P python3`

5.1 Files Needed for Input

In order to sucessfully run Bertini_real, the program needs to be able to access the original `input` file that was used in Bertini, as well as the `witness_data` file generated by Bertini.

5.2 Command prompt, options

There are a number of inline commands that can be used while running Bertini_real. Below is a table that describes these options:[1]

Table 1: Bertini_real command line options

Option	Alter	Command Line	Description
`-component`	integer index of the component	`bertini_real -component 1`	Decomposes only one component of the entire figure
`-debug`	n/a	`bertini_real -debug`	If used, program will pause for 30 seconds before running for debugging purposes

Continued on next page

[1] This table prepared by Beth Sudkamp. Thanks Beth!

Table 1: *Continued from previous page*

Option	Alter	Command Line	Description
`-dim` or `-d`	target dimension of solution	`bertini_real -d 2`	Sets a target dimension to be used for the solution
`-gammatrick` or `-g`	`1` (if you'd like Bertini_real to use the gamma trick) or `0` (if not)	`bertini_real -g 1`	Indicator for whether Bertini_real should use the gamma trick in a particular solver
`-help` or `-h`	n/a	`bertini_real -h`	Displays a help message containing the version of Bertini_real, where Bertini_real can be found online, support information, and finally the command line options.
`-input` or `-i`	filename	`bertini_real -i myfile`	Used if input file is named something other than 'input'
`-mode` or `-m`	`bertini_real` (default) or `crit`	`bertini_real -m crit`	Sets the mode of Bertini_real to be used
`-nostifle` or `-ns`	n/a	`bertini_real -ns`	If used, screen output will not be stifled
`-nomerge` or `-nm`	n/a	`bertini_real -nm`	Indicates that Bertini_real should not merge ends
`-output` or `-out` or `-o`	name of the output directory	`bertini_real -out bertinir_results`	Places the output files in a different directory

Continued on next page

Table 1: *Continued from previous page*

Option	Alter	Command Line	Description
`-projection` or `-pi` or `-p`	desired filename	`bertini_real -p myprojection`	Indicator for whether to read the projection from a file, rather than randomly choose it
-pycommand or `-P`	how to call the Python3 library on your machine	`bertini_real -P python3`	Indicate how Python should be called. Default is 'python'
`-quick` or `-q`	n/a	`bertini_real -q`	Solves problem quickly, but not as robust
`-veryquick` or `-vq`	n/a	`bertini_real -vq`	Solves problem very quickly, but not as robust
`-sphere` or `-s`	the name of the file for Bertini_real to read	`bertini_real -sphere mysphere`	Sets indicator that Bertini_real should use sphere created by user rather than just compute sphere
-symengine or `-E`	choose the engine to run bertini_real	`bertini_real -E python`	select a symbolic engine. choices are 'matlab' and 'python'
`-verb`	the level of the verbosity	`bertini_real -verb 2`	Shows or hides output text
`-version` or `-v`	n/a	`bertini_real -version`	Displays the version of Bertini_real running on your computer

5.3 Parallelism

Bertini_real is parallel-enabled, using MPI (but not OpenMP or threads). To use multiple processors, call it as you would any other MPI program: `mpiexec [mpioptions] bertini_real [broptions]`.

5.4 Projections and spheres of interest

Here we describe the `pi` file in Section 5.4.1, and a `sphere` of interest in Section 5.4.2.

5.4.1 The user-defined projection, `pi`

The `pi` file, defining a specific projection to use for decomposing your curve or surface, has a simple format. You indicate the number of variables, and then give the projection. No punctuation or delimiters necessary.

Using a particular projection, contained in a file of arbitrary name, is indicated to Bertini_real by passing the `-pi` flag. For example, `bertini_real -pi my_projection`.

By default, Bertini_real uses a randomly generated projection to decompose the object. This is so that the object is in *general position*, which is required for set-of-measure-zero guarantees that all elements of the critical space lie in the distinct fibers of the projection.

For decomposing surfaces, if you feel the need to supply your own projection, please consider using two projections π_1 and π_2 such that $\pi_1 \cdot \pi_2 = 0$. That is, orthogonal projections tend to produce cleaner decompositions.

```
num_coords   <-- the number of variables in the problem

pi_1_1  <-- curves and surfaces need at least one projection
pi_1_2
...
pi_1_N

pi_2_1   <--  only surfaces need a second projection
pi_2_2
...
pi_2_N   <-- as many entries in each projection as there are
    coordinates
```

File 9: A file describing a user-defined projection used to decompose a real object in 4 dimensions. The first number indicates the number of coordinates, which must match the number in the object's ambient space. Then, the values for the projection. Try to use orthogonal projections for surfaces.

5.4.2 The sphere of interest, `sphere`

While Bertini_real will happily compute a bounding sphere for you, containing all the interesting parts of your object, it may be very large, or kind of wonky in the case of some projections. Hence, we allow the user to specify their sphere of interest by way of plain text file.

The `sphere` file allows the user to bound the space in which to decompose their object. If there is a region of space you are interested in, you can compute your object inside a sphere of interest.

To inform Bertini_real that you are using your own sphere rather than the computed one, use the `-sphere` flag. For example, `bertini_real -sphere my_sphere_file`. This generally will not speed

up computation at all, since there's no way to know prior to point computation whether the endpoint will be in or out of the sphere. All it will change is the bounded region.

```
radius

x_1_center
x_2_center
...
x_N_center
```

File 10: A file describing a sphere of interest to Bertini_real. The radius appears first, followed by the coordinates of the center of the sphere. Real coordinates only, omit the imaginary part.

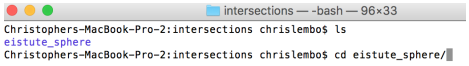
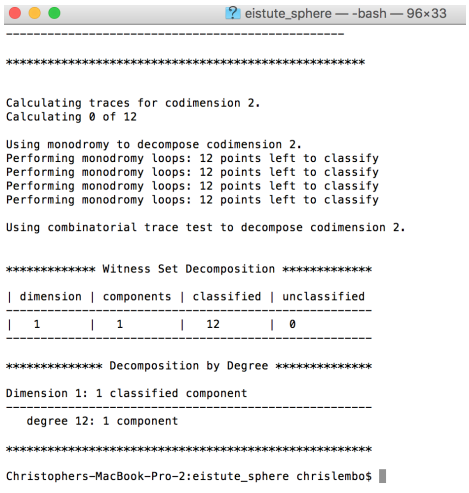
6 Running Sampler

- If you are happy with the results of the Bertini_real decomposition, you may wish to refine the triangulation of the surface or curve. This can be acheived using the `sampler` program after calling `bertini_real`. Sampler can be used on both curves and surfaces.
- This section will show you how to:
  1. Properly run sampler, with visual examples
  2. Use the different algorithms to shape curves and surfaces
  3. Use matlab to better visualize curves and surfaces

6.1 Curves

6.1.1 Running Sampler (Using an Example)

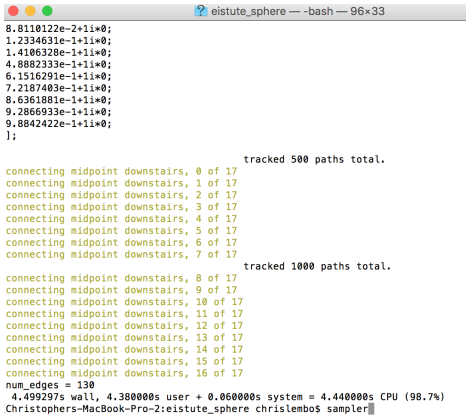
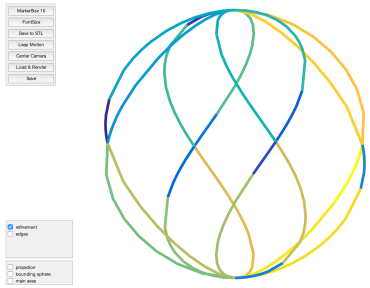
In order to show how to properly run sampler, I will be using an example of a curve, going through each step to make sure the basics of sampler are covered.[2]

Instructions	Screen Shot
First, choose the curve you wish to produce. (In this case I am choosing the `eistute_sphere`, which is found in the `intersections` file which can be found in the `zoo` file)	
Invoke `bertini` and `bertini_real` by entering in each on the command line individually	

Continued on next page

[2] This content prepared by Chris Lembo. Thanks Chris!

Table 2: *Continued from previous page*

Instructions	Screen Shot
Invoke `sampler` by entering it in on the command line	
Now use Matlab to produce the image of the curve 1. Go to the folder that holds your curve, then type `gather_br_samples` 2. To produce the image, type in `bertini_real_plotter` 3. you should end up with a figure along with matlab's display of viewing options	

6.1.2 Available curve sampling algorithms

There are three curve sampling algorithms available in the Sampler module for Bertini_real.

1. `-m [a]` – Adaptive – by movement of a predicted point. Default choice.
2. `-m d` – Adaptive – by distance between consective samples
3. `-m f` – Fixed – every edge gets the same number of points

The modes are accessed by calling sampler with the appropriate mode switch.

Curve sampling algorithm: Adaptive by movement This algorithm refines edge-by-edge until the predicted point, and the computed point, are close to each other. It reduces sample density in regions of the curve which are flat, and has higher density in regions of high curvature.

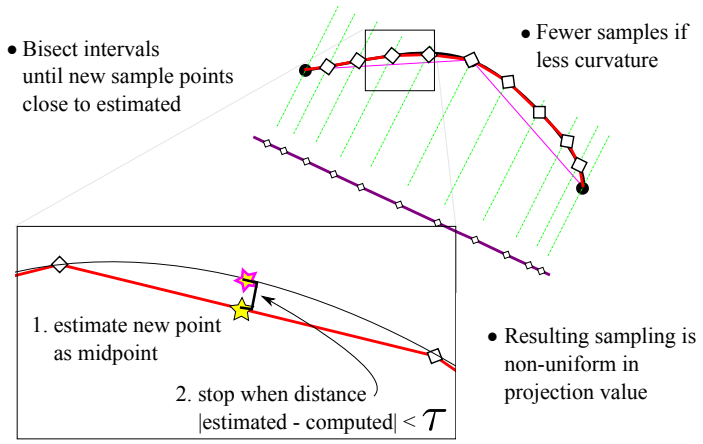


Figure 1: Sampling a curve using movement-adaptive method. New points are computed on the curve by bisecting intervals until the distance between the estimated midpoint and actual midpoint is less than convergence threshold τ.

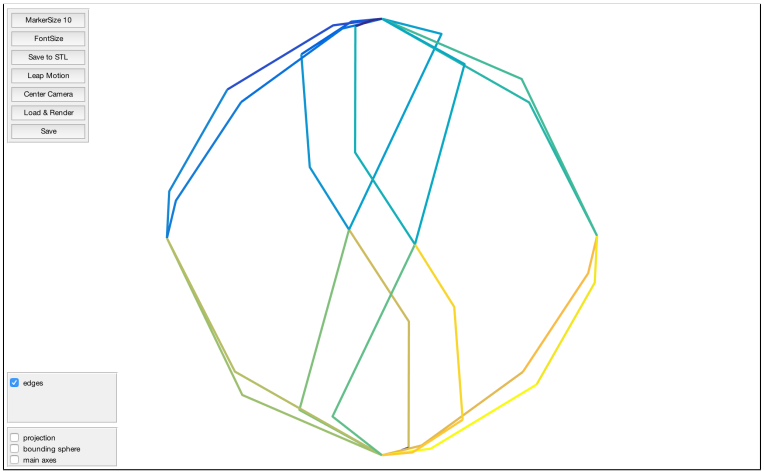


Figure 2: This is an example of sampling a curve using the adaptive by movement mode. To replicate this, when invoking sampler, type in `sampler -m a`

Curve sampling algorithm: Adaptive by distance The adaptive-by-distance refinement algorithm refines the edge by bisection until the distance between consecutive samples is less than some user-set threshold, or a maximum number of refinement iterations has been computed. This algorithm works well, but over-samples flat regions of the curve, where relatively few points should be necessary.

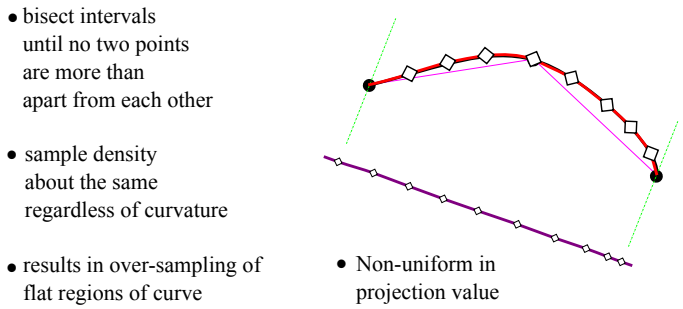


Figure 3: Sampling a curve using distance-adaptive method. New points are computed on the curve by bisecting intervals for which the distance between consecutive samples is larger than the convergence threshold τ.

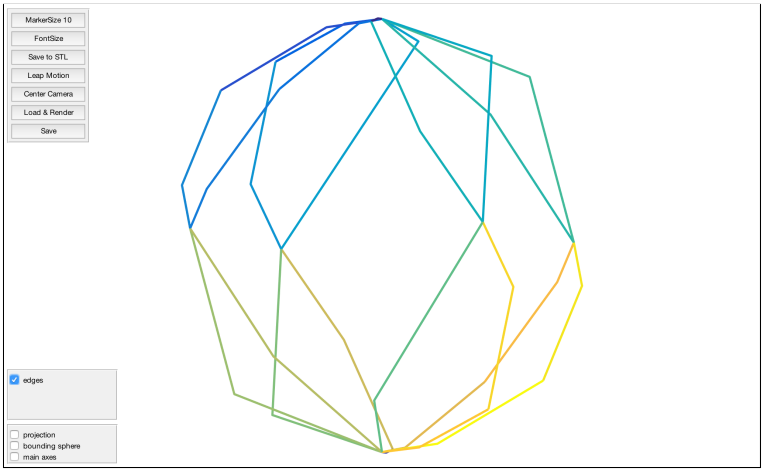


Figure 4: This is an example of sampling a curve using the adaptive by distance mode. To replicate this, when invoking sampler, type in `sampler -m d`

Curve sampling algorithm: Fixed number This curve sampling algorithm produces a fixed (by the user) number of sample points per edge of the decomposed curve.

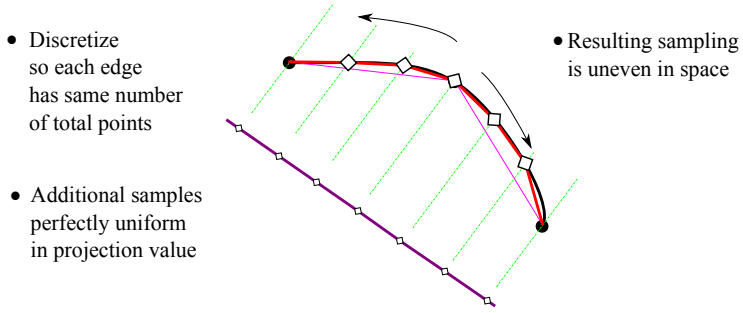


Figure 5: Fixed-number sampling of a curve. The midpoint is homotoped so that a fixed number of sample points are computed on each edge of the curve. The sample points are spaced uniformly in projection value, not in space.

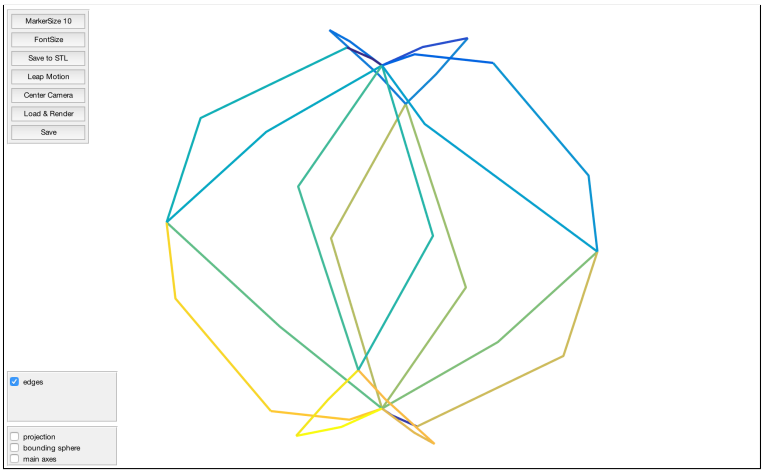


Figure 6: This is an example of sampling a curve using fixed mode. To replicate this, when invoking sampler, type in `sampler -m f`

6.2 Surfaces

6.2.1 Running `sampler` on a surface (Using an Example)

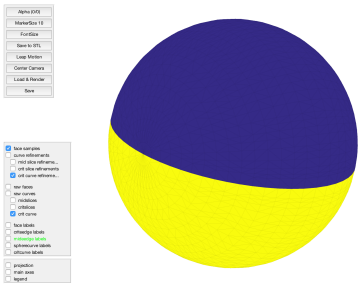
This section will guide you through running sampler using a surface.[3] One thing to note is that it may take longer to invoke the sampler when using it on surfaces due to the amount of computation being done.

Instructions	Screen Shot
First, choose the surface you wish to produce. (In this case I am choosing the `sphere`, which is just found in the surfaces file. Once you enter into this file, you are ready to invoke: • `bertini` • `bertini_real` • `sampler`	surface — -bash — 80×24 input_dim_2_comp_0_deflated witness_data_old input_membership_test witness_points_dehomogenized input_midslice_0 witness_superset input_stabilization_test witness_superset_1 input_surf_sphere Christophers-MacBook-Pro-2:sphere chrislembo$ cd - /Users/chrislembo/code/bertini_real/test/surface Christophers-MacBook-Pro-2:surface chrislembo$ ls 2r eistute seepferdchen 3r fivebar_isotropic solitude barth6 fourd spectahedra buggle hyperboloid_one_sheet sphere calypso klein steiner calyx kummer suss cayley_nodal_cubic leopold tanz chmutov nepali torus clebsch nordstrands_weird tuelle columpius octdong two-torus crixxi paper vis-a-vis cube paraboloid whitney daisy plane zitrus dattel polsterzipf distel riemann Christophers-MacBook-Pro-2:surface chrislembo$ cd sphere

Continued on next page

[3]This content prepared by Chris Lembo. Thanks Chris!

Table 3: *Continued from previous page*

Instructions	Screen Shot
Once sampler has been invoked, go to matlab and follow the same steps layed out for the sampler curve process, by gathering and plotting. Depending on refinements, you will end up with a sphere with fewer or more points. Here are the two matlab commands: • `gather_br_samples` • `bertini_real_plotter`	

6.2.2 Available surface sampling algorithms

There are currently about two surface sampling algorithms, one of which is generally far superior to the other. The inferior method is still maintained for posterity, because it produces interesting patterns on some surfaces, and is useful for instruction.

1. `-m [a]` – Adaptive-movement – Default choice. Will be adaptive on distance moved, once it's implemented. Sorry, we're not there yet on this one.

2. `-m f` – Fixed – every edge of every critical-like curve gets the same number of points, and slices and ribs are sampled distance-adaptively.

3. `-m [d]` – Adaptive-distance – Also default choice, because adaptive-movement collapses to the adaptive-distance method, until movement is implemented. The sampling of critical-like curves are non-uniform, having somewhat-uniformly spaced samplings in terms of projection value across the entire curve. Slices and ribs are distance-adaptively sampled.

In short, the difference between distance-adaptive and fixed sampling is twofold:

- Fixed sampling has the same number of 'ribs' on each face, regardless of size.

- Fixed sampler currently linearly spaces sampled ribs. Adaptive sampling spaces the ribs by cycle number, better approximating regions of the surface coming together at a singularity or critical point.

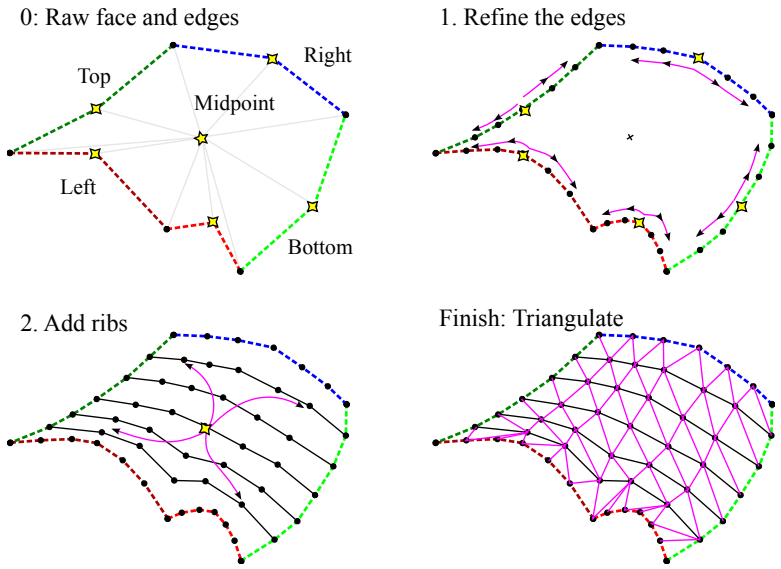


Figure 7: Refining a surface, using the built-in refinement algorithms. Edges of bounding curves are sampled, then the face is ribbed, then the triangulation stitched together. A more familiar adaptive triangulation method from, e.g. computer vision, is not used because this can happen in higher ambient dimensions.

Commonalities The current surface sampling algorithms are two-step methods:

1. Sample the curves using some combination of methods
2. Sample the faces themselves using some method

Current methods also use what Danielle calls 'ribs'. These are curves of samples across the interior of a face, all lying in the same fiber of π_0, so named because they look like ribs on an animal. There are certainly better ways to sample a face. However, the literature almost entirely describes 3D triangulations, and how to optimally form them. Bertini_real can compute triangulations in more coordinates, where many of the necessary ingredients are missing. Research is necessary. Collaborate!

In all surface sampling algorithms, new points in the surface are computed by exploiting the homotopy (6) from Section A.2.4. We will first discuss the weaker of the two methods, because it is briefer and easier to explain.

Surface sampling algorithm: Fixed This surface sampler uses the following parameters:

- `numsamples`
- `tol`
- `minits`
- `maxits`

In the fixed-number surface sampling algorithm,

1. Each edge of each crit-like curve is sampled so it has the same number of points, `numsamples`. The samples are spaced evenly *in terms of projection value*, not space, linearly.

2. Each mid- and critslice is refined using distance-adaptive curve sampler, until `tol` distance is reached, or `min/maxits` refinement iterations are used.

3. The faces are sampled using the distance-adaptive curve method, again using `tol` and `min/maxits`

This method produces mediocre samplings around singularities, because there is too much space between ribs near the challenge points. The adaptive methods are much better at producing a "smooth" sampling, because they use the cycle number of the critical points.

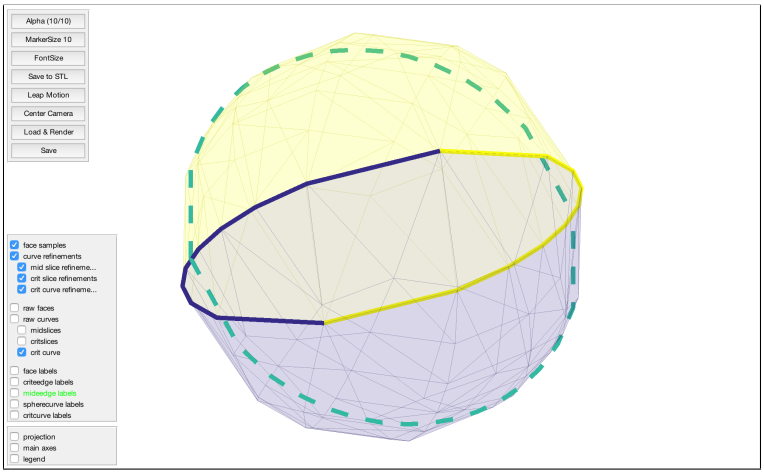


Figure 8: This is an example of sampling a surface using the fixed numsamples mode. To replicate this, when invoking sampler, type in `sampler -m f numsamples 5 -tol 0.7`

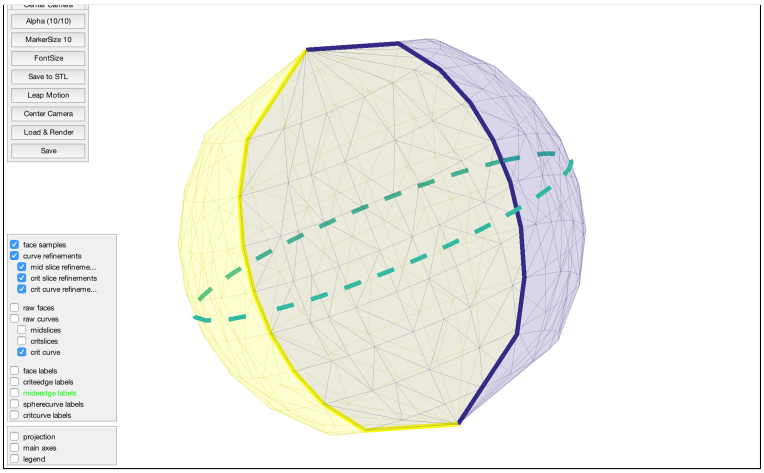


Figure 9: This is an example of sampling a surface using the fixed minits mode. To replicate this, when invoking sampler, type in `sampler -m f minits 5 -tol 0.3`

Surface sampling algorithm: Adaptive by distance This surface sampler uses the following parameters:

- `minribs`
- `maxribs`

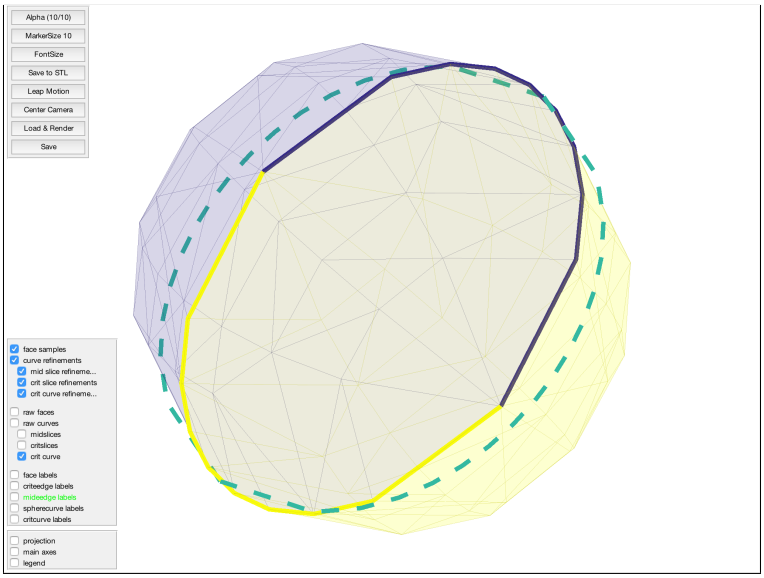


Figure 10: This is an example of sampling a surface using the fixed maxits mode. To replicate this, when invoking sampler, type in `sampler -m f maxits 5 -tol 0.9`

- `minits`
- `maxits`
- `tol`

In the distance-adaptive surface sampler:

1. The crit-like curves are refined using a modified fixed-number sampler. The number of samples per edge is uniform within a particular π_0 crit-interval, but different intervals will have different numbers of samples in them. The number of samples m in an interval is computed as

$$\begin{aligned} m_1 &= \text{max estimated length of crit edge/tol} \\ m_2 &= \min(m_1, \text{maxribs}) \\ m &= \max(m_2, \text{minribs}) \end{aligned}$$

   This means that wider edges will have more samples, narrower will have few.

   We allow the capping of a max number of samples, because small `tol` will lead to insanely large samplings of a complete surface. We also allow a min number of samples, so that even small faces will get refined to some minimal level, which can be important around singularities.

   The samples on the crit-like edges are non-linearly spaced, using the cycle number of the edge to space them. That is, there is a special number, the cycle number c, which can be used to regularize a path. This number is computed as a matter of course while curve decomposing, and hence we store it for use in the sampler routines. Cycle number 2 means, approaching the boundary of an edge, each sample is square root as distant as the previous. Cycle number 3 means it goes as the cube root. Cycle number 1 produces linearly-spaced samples. Ideally the cycle number used to space the samples, and hence ribs on faces, should me an integer multiple of all possible cycle numbers generated by all possible paths on the face, walking toward the boundary.

Presently, the cycle number is forced to be uniform across all edges of the entire surface, because allowing it to vary causes problems when adjoining adjacent faces with disparate cycle numbers. If you want this feature, please consider collaborating with Danielle to add it.

Also, the cycle number is presently hard-coded at 2. Changing this to be a runtime-set parameter value should be pretty easy. Let Danielle know if this is a pressing issue for you.

2. The mid- and critslices are refined using the distance-adaptive curve sampler.

3. The faces are ribbed at the π_0 projection values for the samples on the crit-like edges forming the top and bottom boundaries of the face. The ribbing process is distance-adaptive, with a min and max number of allowable refinement iterations (`minits`, `maxits`) to achieve the desired tolerance `tol`.

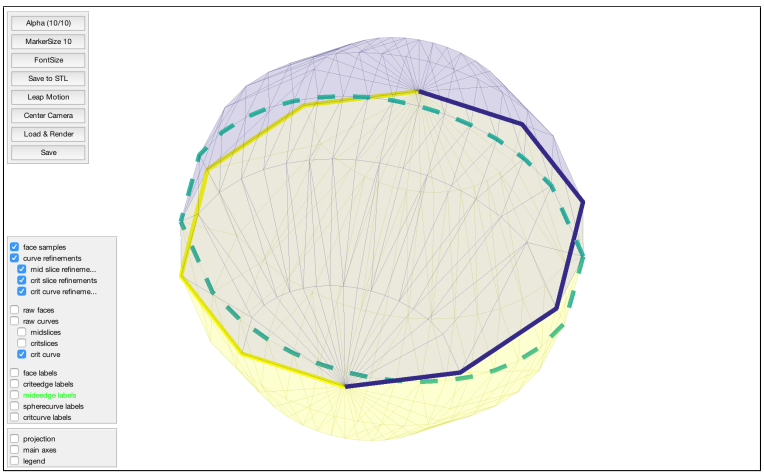


Figure 11: This is an example of sampling a surface using the adaptive by distance minribs mode. To replicate this, when invoking sampler, type in `sampler -m d minribs 10 -tol 0.9`

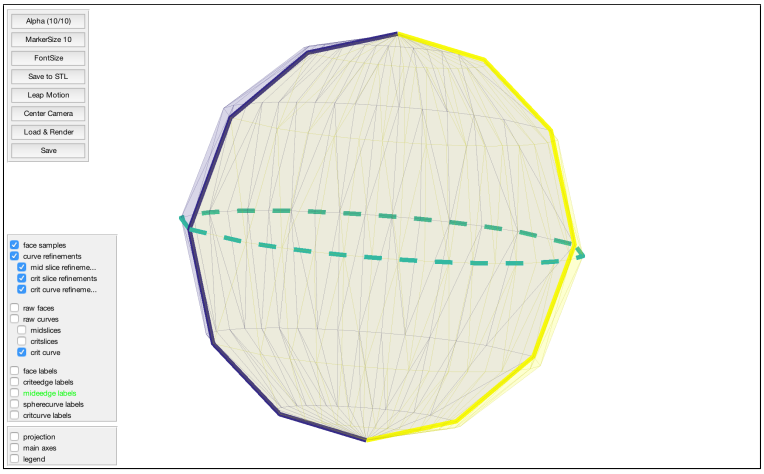


Figure 12: This is an example of sampling a surface using the adaptive by distance maxribs mode. To replicate this, when invoking sampler, type in `sampler -m d maxribs 10 -tol 0.7`

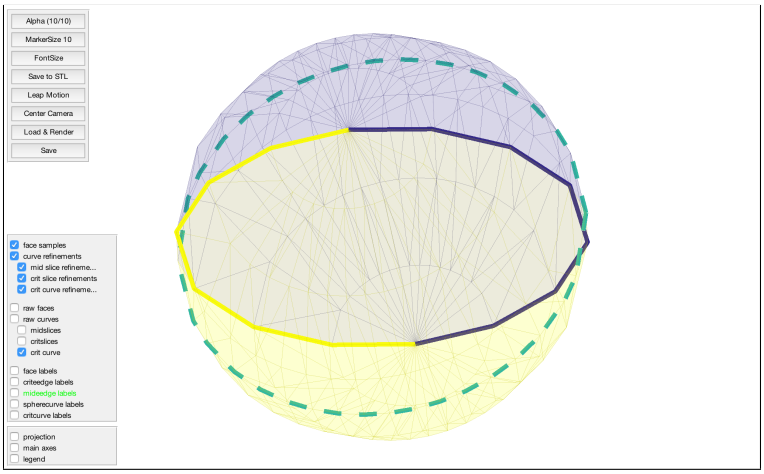


Figure 13: This is an example of sampling a surface using the adaptive by distance minits mode. To replicate this, when invoking sampler, type in `sampler -m d minits 10 -tol 0.5`

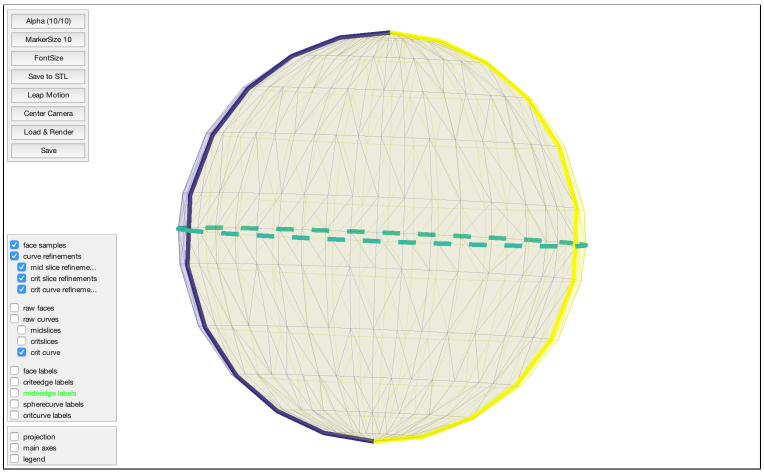


Figure 14: This is an example of sampling a surface using the adaptive by distance maxits mode. To replicate this, when invoking sampler, type in `sampler -m d maxits 10 -tol 0.3`

6.2.3 Known issues with surface sampler

The method for triangulating the faces can sometimes produce janky results. It is known. You'll know it too, when you see it. If you want to help fix this, please contact Danielle. She's ready to collaborate with you!

7 Visualization

7.1 Visualizing in Matlab

After running Bertini_real, the output results can be visualized in Matlab. This section assumes that the Matlab codes for Bertini_real are already on the Matlab path.

1. First, open Matlab, move to the folder in which you decomposed your object, and call `gather_br_samples`. This parses the output from Bertini_real into a .mat file.

2. Then, call `bertini_real_plotter`, which creates a handle class object and facilitates selection of parts of the decomposition to view. There are many options, all of which are documents and displayed via `help bertini_real_plotter` in Matlab.

3. To run bertini_real_plotter with a specific option, type in Matlab `bertini_real_plotter('option', 'option_argument')`, where the option_argument will vary depending on the option you decide to alter. The options are listed below.

7.1.1 Matlab visualization options

Table 4: MATLAB Visualization Options

Option	Default	Alter	Command Line	Description
`'autosave'`	`'on'`	`'false'`, `'0'`	`bertini_real_plotter ('autosave', 'false')` off	Users can automatically save a figure to the working directory or not.
`'colormap'`	`'jet'`	full list here	`bertini_real_plotter ('colormap', @summer)` summer colormap	Users can change the colormap by changing the handle.
`'curve'` or `'curves'`	`'true'`	`'n'`, `'no'`, `'none'`, `'false'`, `'0'`	`bertini_real_plotter ('curve', 'false')` disables the curves option	`bertini_real_plotter`- by default lets the user display the figure's raw curves.
`'faces'`	`'true'`	`'n'`, `'no'`, `'none'`, `'false'`, `'0'`	`bertini_real_plotter ('faces', 'none')` makes only the option to display the raw curves will be given.	By default, the figure created in MATLAB will show both the raw curves and faces.

Continued on next page

Table 4: *Continued from previous page*

Option	Default	Alter	Command Line	Description
'filename' or 'file'			bertini_real_plotter ('filename', 'Example_File_Name.mat')	bertini_real_plotter- first searches files named BRinfo*.mat; if more than one, uses most recent
'labels'	'on'	'n','no', 'none', 'false', '0'	bertini_real_plotter ('labels', 'none') off	bertini_real_plotter- by default lets user apply labels to the figure.
'linestyle'	'-' (solid line)	line options listed here	bertini_real_plotter ('linestyle', ':')	Used to change the line style of lines in the MATLAB figure.
'monocolor' or 'mono'	'off'	RGB triples listed here	bertini_real_plotter ('mono', 'r') creates a red figure	Used to create a mono-color figure.
'proj'				Use a function handle to pre-process the data, before plotting. Lets you plot arbitrary projections of your data
'vertices' or 'vert'	'on'	'n', 'no', 'none', 'false', '0'	bertini_real_plotter ('vertices', 0) off	MATLAB can allow the user to place vertex markers and labels on the figure.

7.2 Visualizing in Python

This portion of visualization code is under active development. There is some helper code currently available in `bertini_real/python/bertini_real`. Add the folder `bertini_real/python` to your

Python path `$PYTHONPATH` environment variable to access it from your Python environment.

7.2.1 Visualizing with Glumpy

Visualization of surface samples is currently supported with Python using Glumpy. Example code can be found at the following webpage: [insert webpage here]

7.3 3D Printing

We realized some time ago that the decomposition of surfaces produces triangulations, suitable for 3d printing with some post-processing. Here's a rough overview of this:

1. Convert data from plaintext to stereolithography (stereolithography (STL)) file. This may involve a projection
2. If necessary, re-orient normals and solidify
3. Process in your favorite model repair service to resolve any geometry problems you may have created in solidifying
4. Print

Danielle has been successful in printing a number of algebraic surfaces, including those either compact and unbounded, those which are everywhere smooth, those having cusp singularity points, and even those with singular curves.

For examples of these, and advice on how to print surfaces, please take a gander at Danielle's online gallery.

8 Troubleshooting

8.1 Helpful help

8.1.1 Compilation fails, with an error due to send calls for MPI

You are probably using OpenMPI, and their implementation does not use `const`-correctness. Danielle needs to modify the code in `bertini_extensions` to cast away constness for these send calls. Send him an email to poke him. Or, use a different implementation of MPI. MPICH2, versions 3.04 and up are known to compile using the code as written.

8.1.2 `had a critical failure`

Missing critical points, or curve slicing problems Sometimes when decomposing an object, Bertini_real will display something like

```
had a critical failure moving left was deficient 2 points trying to recover the failure...
tracktolBEFOREeg:  1e-07 tracktolDURINGeg:  1e-08
```

This is generally due to a missing critical point. Bertini_real uses linear product regeneration to compute critical points, and a missing critical point will cause errors as above.

The solution to this problem is to get Bertini to not miss any critical points. This means understanding the path tracker, and what settings influence tracking success. I generally find several settings can positively influence computation of critical points:

- `securitymaxnorm`, `securitylevel`

  This is the norm of path truncation during the endgame, during path tracking in Bertini. If two successive approximations of points on a path exceed this norm, the path will be truncated, unless `securitylevel` is set to `1`. Setting the security level to 1 will make tracking take longer, as all paths which end at ∞ are tracked all the way to the end. So level 0 spares computation. However, during computation of critical points, synthetic variables representing nullspaces of Jacobians are used, and these can have large norms, resulting in truncation of paths which we need to succeed to fully decompose the object.

  Hence, we recommend setting securitymaxnorm to something large but not crazy large. This is naturally problem dependent. Somewhere around 1e14 has proven useful in our experiments. YMMV.

  To prevent any paths from being truncated, use `securitylevel 1`.

- `condnumthreshold`

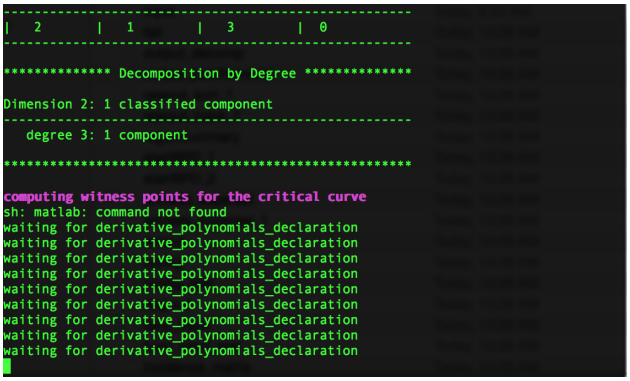
  The post-processor in Bertini classifies endpoints of paths as singular on several criteria, including multiplicity, and on condition number of the Jacobian. To prevent classification of endpoints as singular, raise this threshold. Sometimes large values are needed, upwards of perhaps 1e30 or beyond.

- `sharpendigits`

  After tracking, for nonsingular endpoints Newton's method can be run to increase the accuracy of approximations. This setting sets the number of digits you wish for. You are not guaranteed this number, because numerical conditioning may prevent sharpening from completing.

8.1.3 `sh: matlab: command not found`

If you get the message `sh: matlab: command not found` from running Bertini_real, then Matlab is not on your shell path. Bertini_real currently requires Matlab to run properly, and thus failure to include the Matlab executable on the path will cause bertini_real to fail. Below is an example of the terminal output displayed when Bertini_real is unable to locate the Matlab executable:



Permanently fixing this error involves editing your profile, e.g. `bash_profile`. From your home directory, open `.bash_profile` and add the following line: `export PATH=/PATH/TO/MATLAB.app/bin:$PATH` where `PATH/TO/MATLAB.app` points to the location of the Matlab executable. If you do not have, type `touch .bash_profile`, which will create `.bash_profile`, and then add `export PATH=/PATH/TO/MATLAB.app/bin:$PATH`. This should result in the Matlab executable being added to the path whenever opening terminal.

8.1.4 Calling Bertini_real from within Matlab

Note that you cannot run Bertini_real from within Matlab, even with the use of `!`, because Bertini_real currently calls Matlab using a `system()` call.

Removing dependency on Matlab's symbolic toolbox

Removal of Matlab as a dependency is ongoing work. Please consider helping me move to Python or another open source language for the symbolic computations, required for deflation of singular curves, and the writing of the input file for critical curves of surfaces.

9 Examples

About: This section will run the user through some examples of what this program is capable of along with some higher level discussion about what is going on in each step. Note that these examples are here to demonstrate bertini and bertini_real using the instructions prior to this, so there will not be any step by step analysis here.

9.1 Curves

9.1.1 Circle

Input file

```
CONFIG

tracktype: 1;
mptype: 2;
END;

INPUT

variable_group x,y;
function f1;
f1=x^2+y^2-1;

END;
```

File 11: Presents an input file from circle that instructs Bertini to use all default settings to compute the numerical irreducible decomposition of the sphere in two dimensions

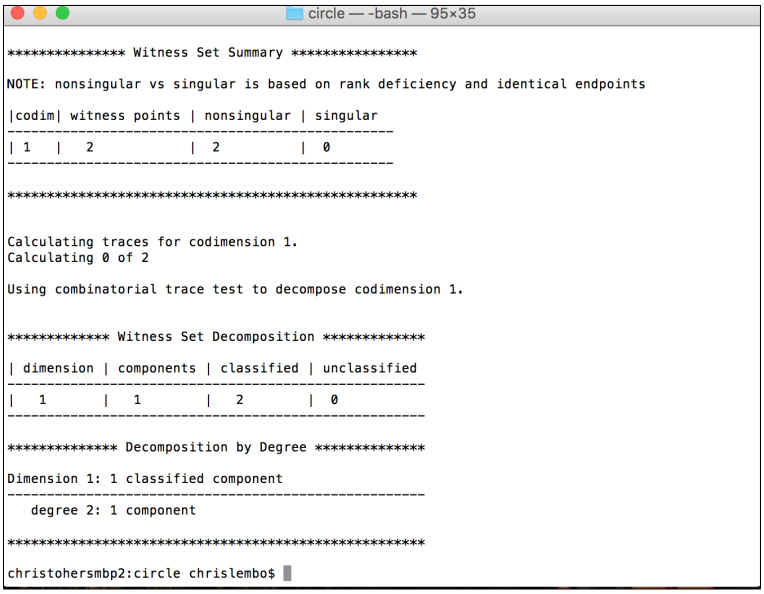


Figure 15: Running Bertini using the Circle input file

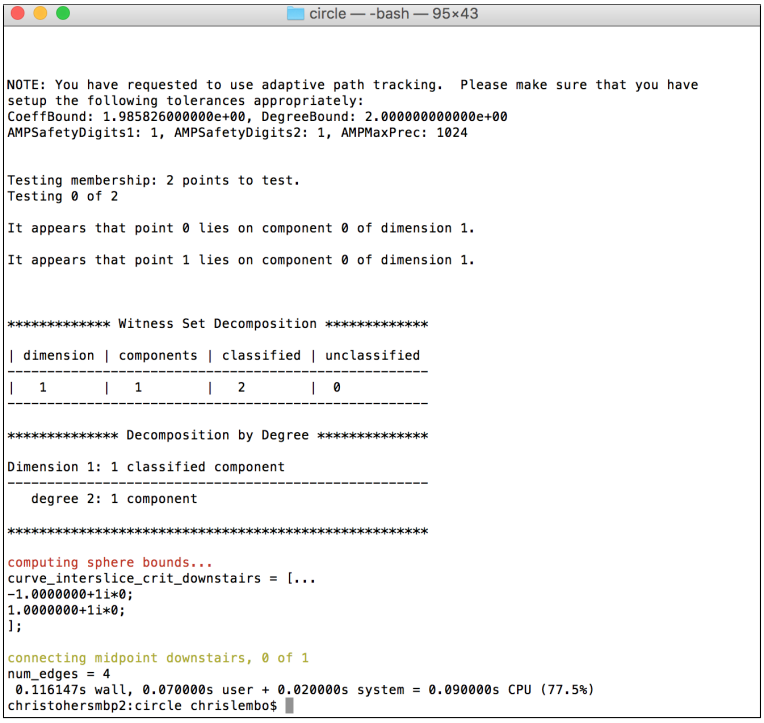


Figure 16: Running Bertini_real using the Circle input file

Decomposition

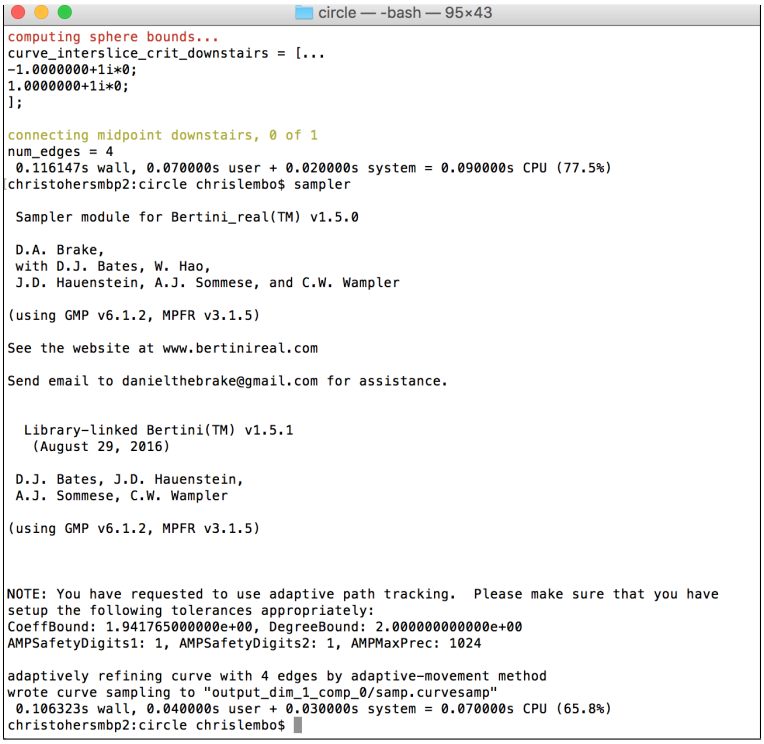


Figure 17: Refining the Circle input file by invoking sampler

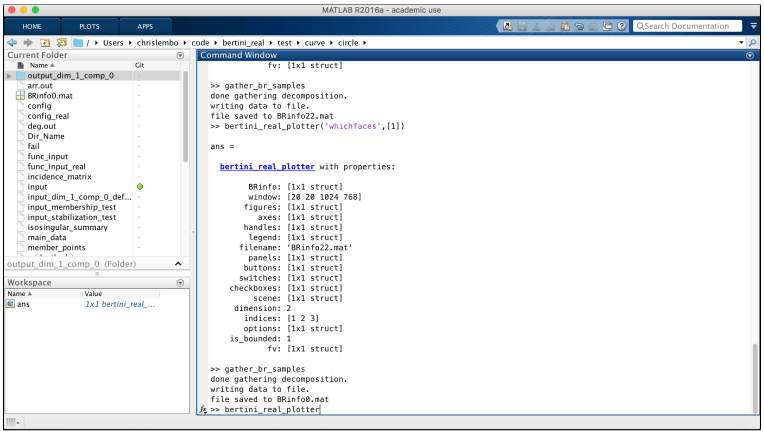


Figure 18: Gathering and Plotting using MATLAB

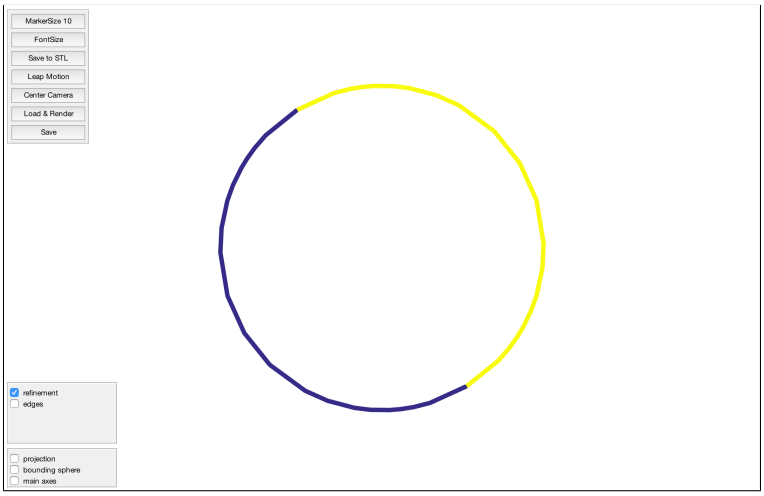


Figure 19: Final Circle plotted

Refinement

Notes

9.1.2 Astroid

Input file

```
CONFIG

tracktype:1;

imagthreshold: 1e-5;
END;



INPUT

variable_group X1, X2;

function f1;

f1 = (X1^2 + X2^2 -1)^3 + 27*X1^2*X2^2;

END;
```

File 12: Presents an input file from astroid that instructs Bertini to use all default settings to compute the numerical irreducible decomposition of the sphere in one dimension

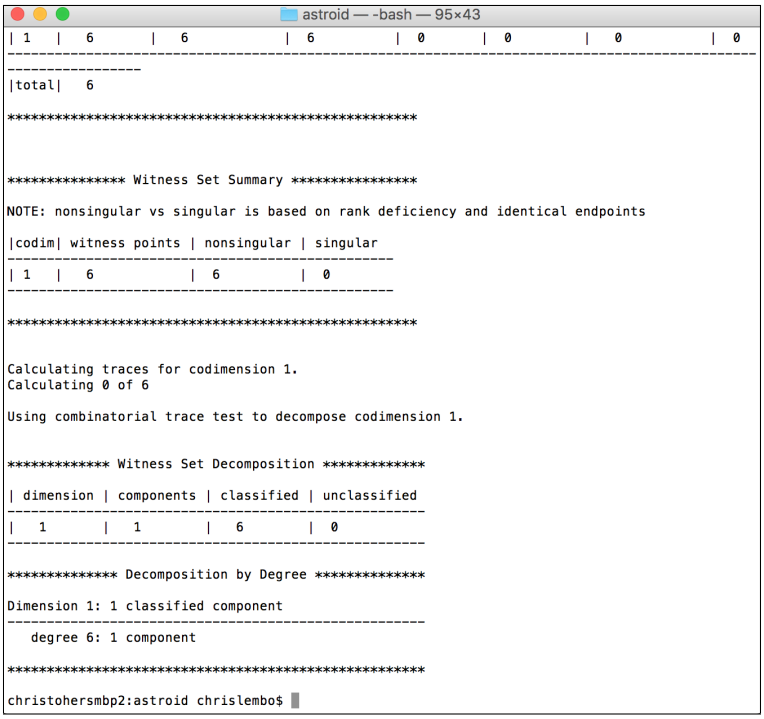


Figure 20: Running Bertini using the astroid input file

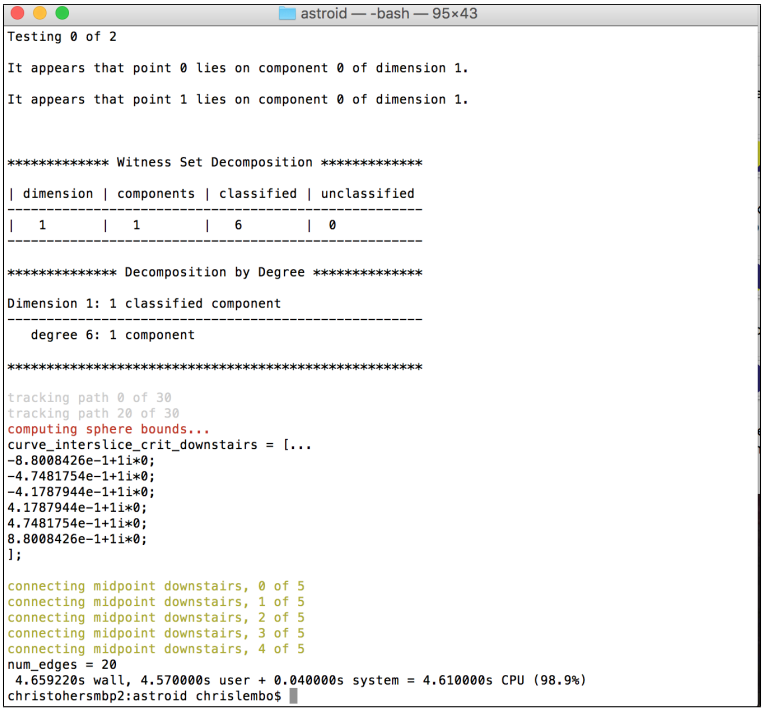


Figure 21: Running Bertini_real using the astroid input file

Decomposition

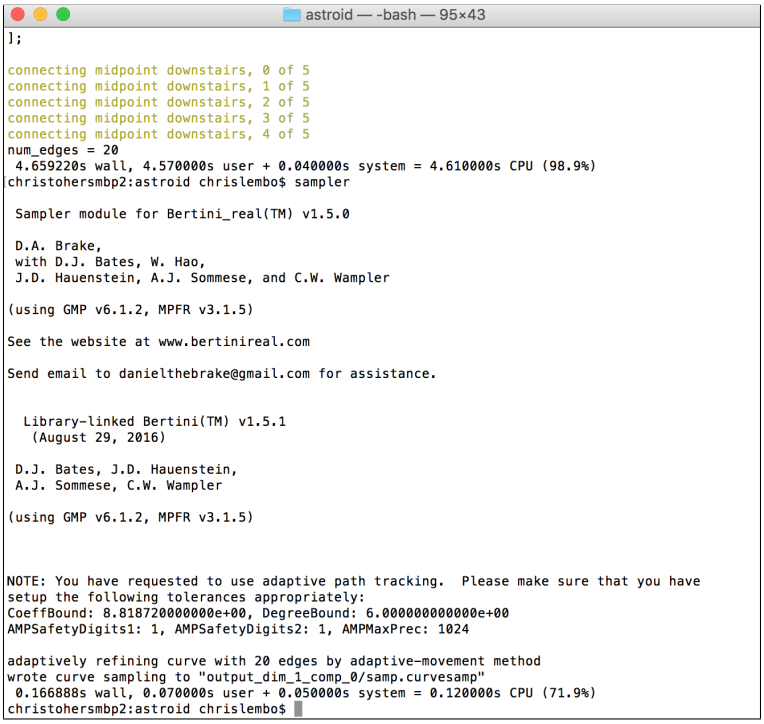


Figure 22: Refining the astroid input file by invoking sampler

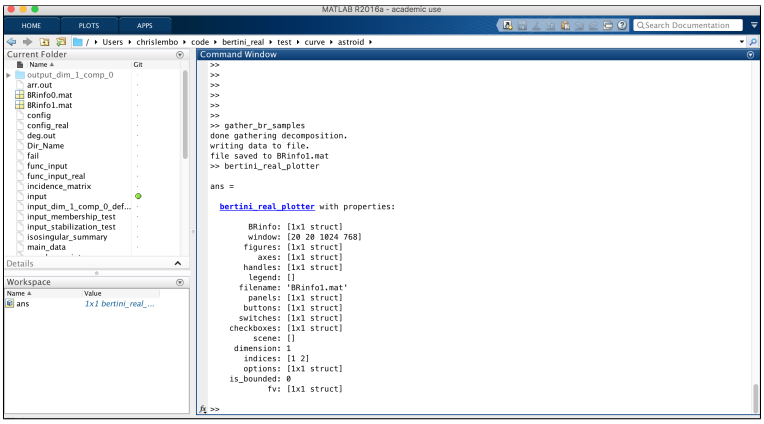


Figure 23: Gathering and Plotting using MATLAB

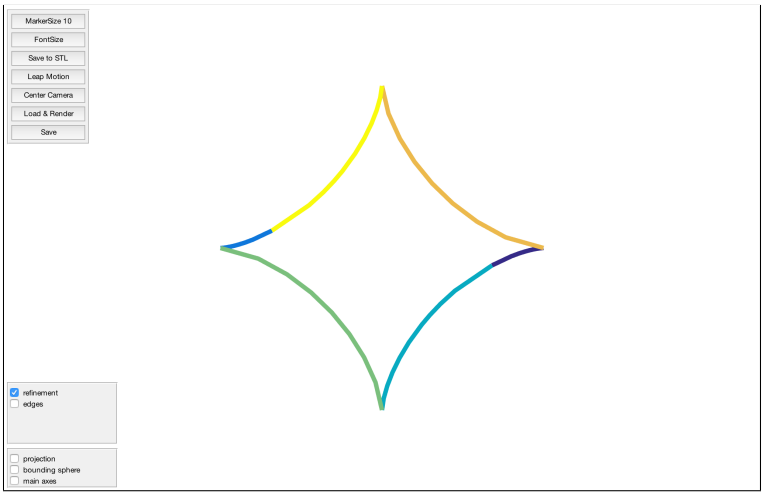


Figure 24: Final astroid plotted

Refinement

Notes

9.2 Surfaces

9.2.1 Solitude

Input file

```
CONFIG
randomseed: 2;
tracktolbeforeeg: 1e-7;
tracktolduringeg: 1e-8;

endgamebdry: 0.001;

endgamenum: 2;
numsamplepoints: 10;

odepredictor: 8;
tracktype:1;
endpointfinitethreshold: 1e10;

securitymaxnorm: 1e16;
condnumthreshold: 1e160;
SINGVALZEROTOL: 1e-140;
sharpendigits: 24;

maxnewtonits: 1;
maxstepsbeforenewton: 0;
END;

INPUT
variable_group x, y, z;
function f;
f = x^2*y*z +x*y^2+y^3+y^3*z-x^2*z^2;


END;
```

File 13: Presents an input file from solitude that instructs Bertini to use all default settings to compute the numerical irreducible decomposition of the sphere in two dimensions

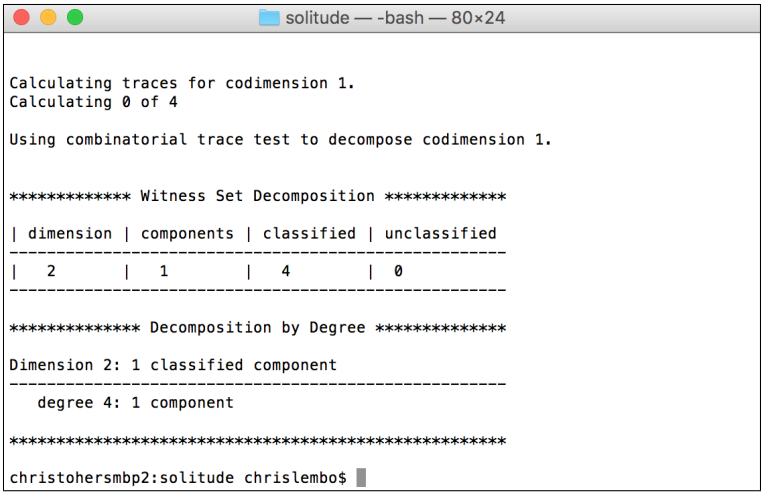


Figure 25: Running Bertini using the solitude input file

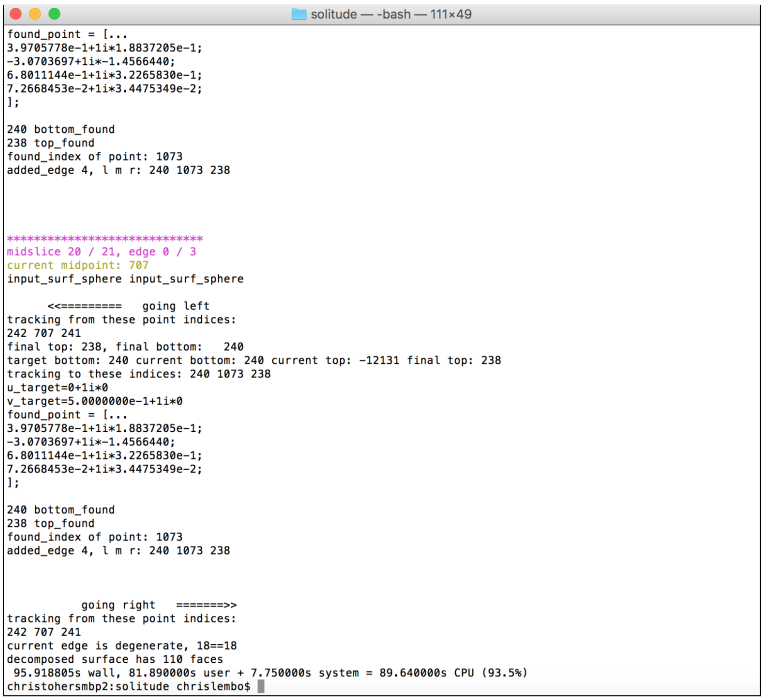


Figure 26: Running Bertini_real using the solitude input file

Decomposition

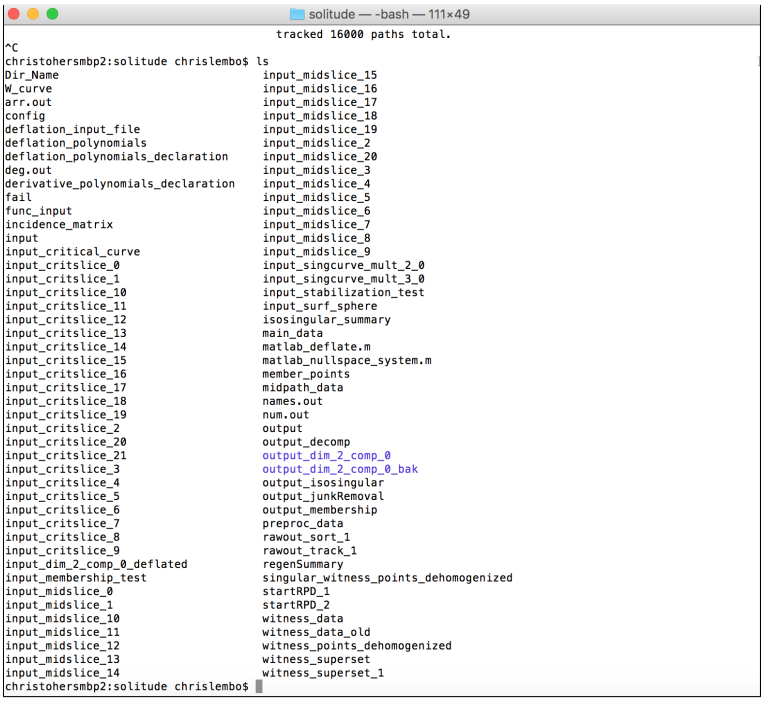


Figure 27: Refining the solitude input file by invoking sampler using adaptive by distance

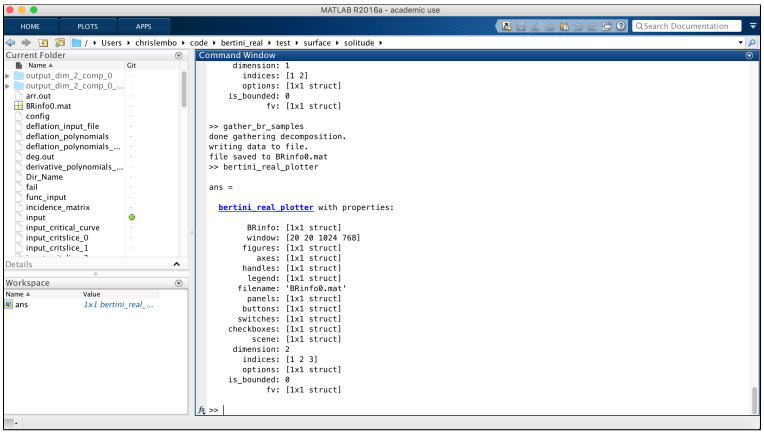


Figure 28: Gathering and Plotting using MATLAB

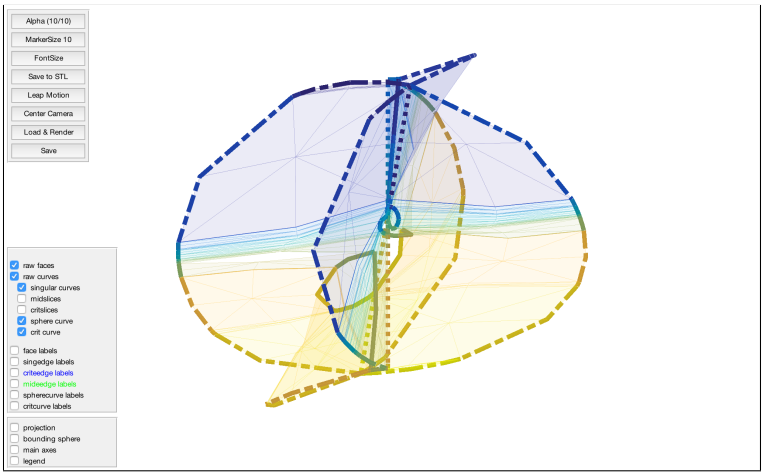


Figure 29: Final solitude plotted

Refinement

Notes

9.2.2 Plane

Input file

```
CONFIG
tracktype: 1;

END;

INPUT
variable_group x, y, z;
function f;
f = x+y+z-1;

END;
```

File 14: Presents an input file from plane(note to make this in code writing) that instructs Bertini to use all default settings to compute the numerical irreducible decomposition of the sphere in two dimensions

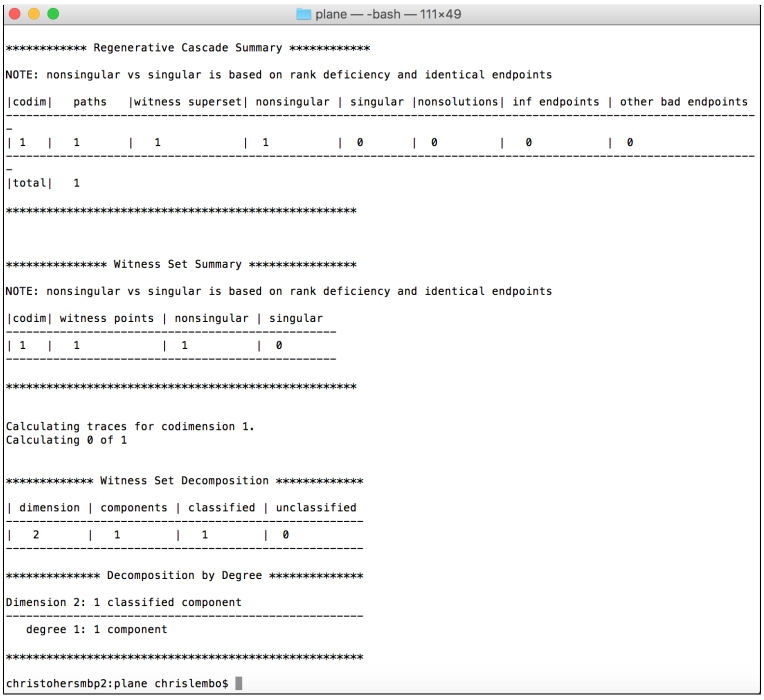


Figure 30: Running Bertini using the plane input file

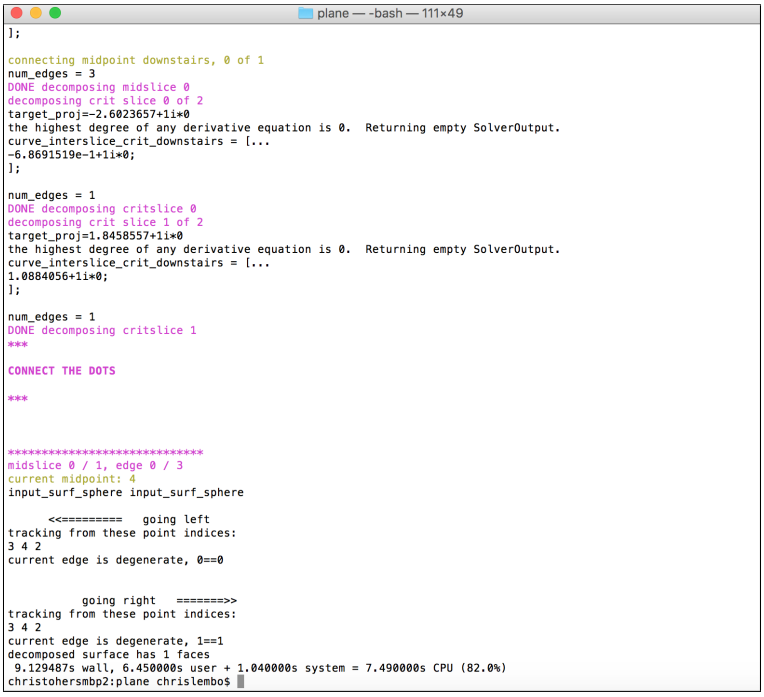


Figure 31: Running Bertini_real using the plane input file

Decomposition

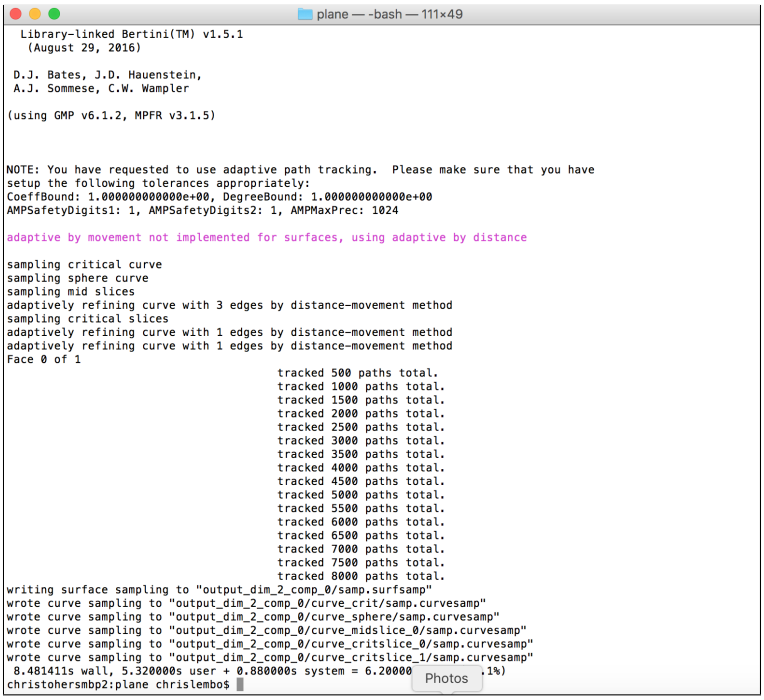


Figure 32: Refining the plane input file by invoking sampler

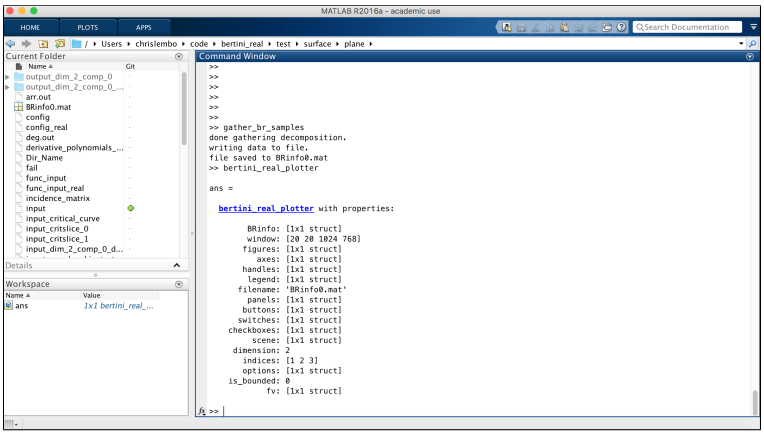


Figure 33: Gathering and Plotting using MATLAB

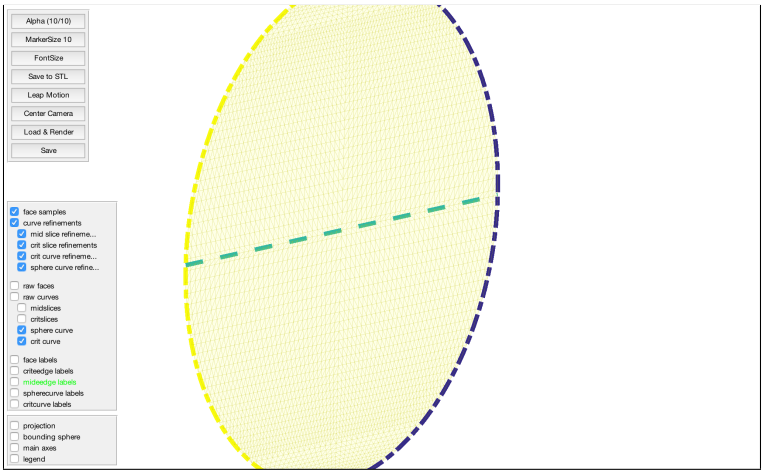


Figure 34: Final plane plotted

Refinement

Notes

References

[1] Dan Bates and Jon Hauenstein. *Bertini Manual*, 2013.

[2] D.J. Bates, J.D. Hauenstein, A.J. Sommese, and C.W. Wampler. *Numerically solving polynomial systems with Bertini*, volume 25. SIAM, 2013.

[3] G.M. Besana, S. Di Rocco, J.D. Hauenstein, A.J. Sommese, and C.W. Wampler. Cell decomposition of almost smooth real algebraic surfaces. *Numerical Algorithms*, 63(4):645–678, 2013.

[4] Danielle Brake, Daniel Bates, Wenrui Hao, Jonathan Hauenstein, Andrew Sommese, and Charles Wampler. On computing a cell decomposition of a real surface containing infinitely many singularities. 2014.

[5] Danielle Brake, Daniel Bates, Wenrui Hao, Jonathan Hauenstein, Andrew Sommese, and Charles Wampler. Algorithm xxx: Bertini_real: Numerical decomposition of real algebraic curves and surfaces. February 2015.

[6] Danielle A. Brake, Daniel J. Bates, Wenrui Hao, Jonathan D. Hauenstein, Andrew J. Sommese, and Charles W. Wampler. *Mathematical Software – ICMS 2014: 4th International Congress, Seoul, South Korea, August 5-9, 2014. Proceedings*, chapter Bertini_real: Software for One- and Two-Dimensional Real Algebraic Sets, pages 175–182. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[7] J.D. Hauenstein and C.W. Wampler. Isosingular sets and deflation. *Foundations of Computational Mathematics*, 13(3):371–403, 2013.

[8] Y. Lu, D.J. Bates, A.J. Sommese, and C.W. Wampler. Finding all real points of a complex curve. *Contemporary Mathematics*, 448:183–205, 2007.

[9] MartyMacGyver. How to install a newer version of gcc, July 2011.

A The Decomposition Algorithms

This section of the manual describes, hopefully without too much techinical detail, the curve and surface decomposition algorithms. The main academic paper on this is [3], while the paper on Bertini_real implementing it is [5]. Two additional extended abstracts discussing it are [6, 4].

We invite you to play with the software and visualization routines, to experience these algorithms first-hand. Danielle in particular thinks of these algorithms as implementations of the implicit function theorem. Enjoy!

A.1 Decomposing curves

Decomposing an algebraic curve numerically can be summarized easily in six steps:

1. Compute critical points
2. Intersect with bounding object
3. Slice at projection interval midpoints
4. Connect the dots
5. Merge [optional]
6. Sample [optional]

A curve is decomposed with respect to projection onto a (randomly chosen) real linear projection, $\pi_0(x)$.

What does it mean to decompose a curve? To turn compute a set of *edges* that describe the curve. An edge is a 1-dimensional object, having two points as boundary, and a general point in the middle, together with a homotopy which can be used to track the midpoints between the boundary points. Phew. See Figure 35.

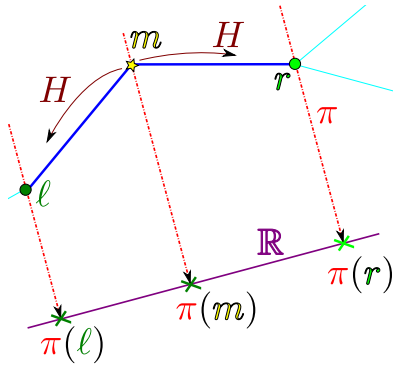


Figure 35: An edge, a 1-cell, as a component of a curve decomposition.

A.1.1 Critical points

Critical points of a curve satisfy the following system:

$$\begin{bmatrix} f(x) \\ \det \begin{pmatrix} Jf(x) \\ J\pi_0 \end{pmatrix} \end{bmatrix} = 0. \tag{1}$$

These points include singular points (trivially, and independently of projection). This is solved by a 2-homogeneous regeneration procedure.

A.1.2 Intersect with bounding object

To capture the behaviour of the curve as it goes to ∞, we intersect the curve with a bounding object, and ignore all outside points. In Bertini_real, we use a sphere centered at the centroid of the critical points, with radius 3 times the distance to the furthest critical point. The user can choose their own sphere of interest.

A.1.3 Slice

Let the set of critical points just computed be χ. Then, take $\pi_0(\chi)$, the set of critical projection values. We slice at midpoints of projection intervals defined by this set. This computes the set of midpoints for (unmerged) edges computed by the next step.

A.1.4 Connect the dots

Connecting the dots for a curve simply means to track each midpoint to the left and right bounding critical projection values, and see what critical points match. This forms an edge. The homotopy

$$(1-t)\begin{bmatrix} f(x) \\ \pi_0(x) - p_{c_j} \end{bmatrix} + t\begin{bmatrix} f(x) \\ \pi_0(x) - p_{m_i} \end{bmatrix} = 0 \tag{2}$$

is used.

Some midpoints may track to points in critical fibers which have not yet been previously computed. In software, these are classified as *semicritical*.

A.1.5 Merge [optional]

Merging edges combines two edges which meet at a non-critical point. These points occur when tracking midpoints toward critical projection values yields a new point, one in the fiber of a true critical point. A simpler curve decomposition can be obtained by merging such edges. There are times when merging is great (most of the time) and times when merging is not the right thing to do (some points in the surface decomposition algorithm).

Merging can be disabled when decomposing a curve in Bertini_real with the `-nomerge` flag.

A.1.6 Sample [optional]

Suffice it to say for now that a midpoint is tracked using (2) within the bounds of its edge, producing additional points on the edge. This section is described in much greater detail in Section 6.1.

A.2 Decomposing surfaces

Decomposing an algebraic surface numerically is similar to that of a curve, and can also be summarized in about six steps:

1. Compute critical curves
2. Intersect with bounding object
3. Slice at projection interval midpoints. The slices are curve decompositions
4. Connect the dots to form faces
5. Merge [optional]
6. Sample [optional]

To decompose a surface is to compute a set of *faces*. A face is a 2-cell, containing a set of bounding edges, and a general point in the middle, which can be tracked using a homotopy, staying within the boundary. See Figure 36

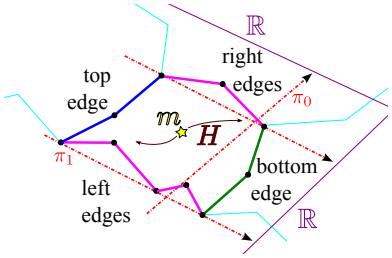


Figure 36: A face, a 2-cell, as a component of a surface decomposition. Its boundary consists of edges from curve decompositions.

A surface is decomposed with respect to projection onto a pair of (randomly chosen) real linear projections, $\pi_0(x)$ and $\pi_1(x)$, together which may be referred to as $\pi(x)$.

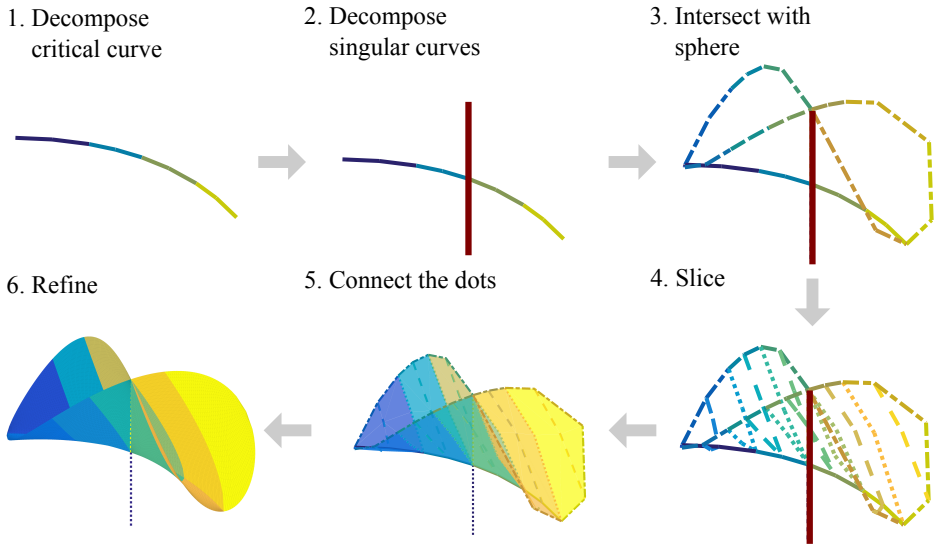


Figure 37: Numerical cellular decomposition of the Whitney Umbrella using Bertini_real.

A.2.1 Critical curves

Critical curves can be separated into two categories. First, those which are an artifact of the projection being used to decompose. Second, those which appear regardless of the projection. In this manual and software, we refer to the former simply as the *non-singular critical curve* or simply *critical curve*, and the second as *singular curves*, although they both are formally part of the critical curve (and so is the bounding curve).

Nonsingular critical curve The non-singular critical curve satisfies this system, nonsingularly:

$$F(x) = \begin{bmatrix} f(x) \\ \det \begin{pmatrix} Jf(x) \\ J\pi_0 \\ J\pi_1 \end{pmatrix} \end{bmatrix} = 0. \tag{3}$$

In current implementation using Bertini1 as the homotopy engine, we use a symbolic engine to compute a new text Bertini input file. There are several options for the symengine, described in Section 5. A curve decomposition is run on this curve. But first, we have to have **all** of the critical points of all critical curves, including non-singular, singular, and bounding.

Singular curves Not every surface has singular curves, but many do. Perhaps the easiest to see at first is the Whitney Umbrella.

$$f = x^2 - y^2 z = 0 \tag{4}$$

This equation describes a degree 3 surface. The z-axis is singular. Observe the Jacobian:

$$Jf = \begin{bmatrix} 2x & -2yz & -y^2 \end{bmatrix} \tag{5}$$

On the z-axis, $x = y = 0$. So, Jf is singular. Hence the determinant in (3) is 0, which means the critical curve system is in some sense trivially satisfied. What is needed here is to *deflate* the system. This is accomplished in current implementation using *isosingular deflation* [7]. Basically, sub-determinants of the system are recursively added until the rank of the Jacobian stabilizes, where this rank is computed using a witness point for the component being deflated.

The singular curves are decomposed using the curve algorithm, after obtaining *all* critical points of *all* critical-like curves.

A.2.2 Bounding curve

To capture the behaviour of surfaces which are not closed or bounded, we intersect the surface with a bounding object. The implemented type in Bertini_real is sphere. Initially, we had implemented a bounding box, and it was neat, but for higher dimensions, it meant doing more and more curve decompositions, with each of the $2N$ planes. It got messy. So, surfaces are used. They are easier, because the entire bounding curve satisfies a single system, which is merely the original system supplemented with a single degree 2 equation – that of the sphere. It eases not only implementation but also runtime.

We compute the sphere automatically, after having computed the critical points of the critical and singular curves (but before decomposing them, for algorithmic reasons). The sphere is centered at the centoid of all critical points, and its radius is arbitrarily chosen to be 3 times the distance from the center to the furthest critical point. The user may also supply their own sphere of interest (See Section 5).

A.2.3 Slice

Having decomposed all formal parts of the critical curve, we slice the surface. Where?

Consider all critical points of all thus-far computed curves. Call this set χ. Then we project onto the first projection coordinate, computing $\pi_0(\chi)$. This set of values produces the critical slices. Taking midpoints of all intervals defined by $\pi_0(\chi)$ defines the mid slices. Each of these slices is computed using a regular curve decomposition.

A.2.4 Connect the dots

This is a fun piece of the puzzle. In this part of the algorithm, midpoints of edges of midslices are connected to midpoints of edges of critslices. We track the mid-midpoints using a very special homotopy, the *midtrack* homotopy:

$$\begin{bmatrix} f(x) \\ \pi_0(x) - [(1-u)\pi_0(c_i) + u\pi_0(c_{i+1})] \\ f_{\text{bottom}}(y) \\ \pi_0(y) - [(1-u)\pi_0(c_i) + u\pi_0(c_{i+1})] \\ f_{\text{top}}(z) \\ \pi_0(z) - [(1-u)\pi_0(c_i) + u\pi_0(c_{i+1})] \\ \pi_1(x) - [(1-v)\pi_1(y) + v\pi_1(z)] \end{bmatrix} = 0. \tag{6}$$

The homotopy is somewhat hidden in (6) – as written the t-dependence is implicit. To track the homotopy, we use:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} (1-t)\, u_{\text{target}} + t\, u_{\text{start}} \\ (1-t)\, v_{\text{target}} + t\, v_{\text{start}} \end{bmatrix}. \tag{7}$$

The values of u and v lie in the unit square, and we must only track inside this square. The purpose of this homotopy is to ensure that the midpoint never crosses an edge of the critical curve, which bounds the in-construction face to the top and bottom. That is, if we just did a straight-line homotopy in π_i coordinates, we would likely cross this critical boundary, and all bets would be off. Paths would cross, real points would become complex, and the decomposition would not work. Hence, this homotopy. It's used during sampling, too.

Anywho, the midpoints are connected to midpoints. The midpoints of midslice edges become midpoints of faces in the cell decomposition. Critslice edges bound the face to the left and right (π_0), while critcurve edges bound to the top and bottom (π_1). There may be any number of left and right edges bounding a face, but exactly one top and bottom edge.

A.2.5 Merge [optional]

The algorithm for merging faces does not yet appear in the literature, though it has been described verbally by Charles Wampler. Let's write it! Should be a short paper, probably targeting a conference proceedings. Contact Danielle Brake to co-author it with her.

A.2.6 Sample [optional]

This section is described in much greater detail in Section 6.2. For now, suffice it to say that (6) is used to track around the midpoint of each face, producing additional points on the face, and that a proper triangulation is maintained. There are choices about how to sample. One might think this a trivial problem. Indeed, in three dimensions, there are many many algorithms for adaptively and optimally triangulating surfaces. However, in higher dimensions the normal vector doesn't exist, and cross-products cannot be taken, so much of the machinery from 3d breaks. Bummer. Go check out the section on `sampler` for more.

B Installation

This section of the manual focuses on how to install the necessary dependencies and programs needed to run Bertini_real on a user's computer. The instructions provided describe the process for Linux, Mac, and Windows operating systems. If you need help or encounter an problem, please file an issue on Github at github.com/ofloveandhate/bertini_real/issues. This is preferred over email.

When installing Bertini_real, there are a number of steps required in order to successfully install and run the program. They are:

1. Installing the dependencies
2. Installing Bertini
3. Installing Bertini_real

B.1 Dependencies

Before installing Bertini and then Bertini_real, there are a number of packages that need to be installed. The method used to install these dependencies changes depending on the operating system, so please be sure to read the section that describes your particular system. Except for Bertini, you should be able to install all of these using a package manager. It's there to help.

Bertini dependencies

- Multiple Precision Floating-Point Reliable (MPFR)
- GNU Multiple Precision Arithmetic Library (GMP)
- MPI (whatever implementation you want)
- flex
- bison

Bertini_real dependencies

- a C++ compiler capable of the C++11 standard
- Bertini, **parallel version compiled from source and installed as library**.
- Message Passing Interface (MPI) (Must be the same one as Bertini was compiled with)
- Boost >= 1.53
- `autoconf`, `automake`, `make`, `libtool`. *Please use your package manager.* Also, maybe `pkg-config`. If you hate the autotools, please set up CMake for Bertini_real, and do a pull request. Seriously.

Bertini_real is parallel-enabled, using MPI. **You cannot build Bertini_real without support for MPI at the current time.** To use multiple processors to decompose a real object, call Bertini_real as you would any other MPI program: `mpiexec [options] bertini_real [options]`. It also works in serial, without being hosted by the MPI executor. But you still have to have MPI.

B.1.1 Symbolic engine

One additional piece of software must be installed in order to decompose surfaces – something to do symbolic work for us. Two options are available at this time (Fall 2018): Matlab, and Python. Matlab has much stronger visualization routines for Bertini_real, and some nice options for improving produced Bertini input files.

MATLAB One option for symbolic engine for Bertini_real is Matlab. Instructions on how to install the program are not provided here. However, if you are associated with a university, or a research facility, they probably have download instructions on their technology support website. Ensure you have access to the symbolic toolbox. It's required.

You must add Matlab to the terminal path – you have to be able to type `matlab` into a terminal and have it launch.

If you intend to use Matlab for visualization, you need to add `path/to/bertini_real/matlab_codes` to the path, as well as several folders in there, namely `matlab_codes/brakelab/bertini1` and `matlab_codes/brakelab/rendering`

Python Bertini_real was improved in Fall 2016 to allow you to use Python as the symbolic engine, in alternative to Matlab. This allows for fully free software to do the decompositions. Visualization routines in Python are in progress, with surfaces being implemented 2018-19 academic year. You need the following packages, ideally from `pip`:

- `sympy`
- `scipy`
- `numpy`
- `algopy`
- `mpmath`

and for visualization

- `dill`
- `matplotlib`

B.2 Instructions for GNU/Linux

1. Install dependencies. You are very strongly encouraged to use the package manager provided, such as `apt` or `yum`, to install the dependencies. See Section B.1 for a list of dependencies.

2. Build and install Bertini1, ensuring that you are building the parallel version. This will happen automatically if you install MPI as a dependency. Remember, you must install from source, simply downloading the pre-built executable will not get you the built libraries.

   (a) `./configure`

   (b) `make -j 4`

   (c) `sudo make install`

3. Build and install Bertini_real.

   (a) Clone from repo, `git clone https://github.com/ofloveandhate/bertini_real`.

   (b) `libtoolize`

   (c) `autoreconf -i` if you know how to eliminate these steps by writing a bootstrap file which works on all systems, please contribute by way of a pull request. PR's gladly acccepted to `develop`.

   (d) `./configure`

   (e) `make -j 4`

   (f) `sudo make install`

   (g) eat something delicious, and give thanks for this peaceful day

* If using Matlab for symbolic engine, ensure it is installed an on the path for the command line.

  In bash, `touch ~/.bash_profile`, then edit it, adding `export PATH=/PATH/TO/MATLAB/bin:$PATH`.[4]

* If using Python for symbolic engine, install your favorite version, `pip`, and the necessary modules via `pip`.

If anything went wrong, please file an issue on Github. I want this to be an easy experience.

[4] Note that there are several possible files into which to place this, and this particular method of using `export` is shell specific. The `csh` is a little different, for example. Too lazy to Google it? I gotcha: search on da googs here

B.3 Instructions for OSX

B.3.1 Preliminary Work

- Compilers &c: Install XCode 8 from the App Store. Open it at least once. In terminal, type `xcode-select --install` (you must open XCode at least once before doing this)
- Homebrew: It is highly recommended that Mac users install the program Homebrew to use to install these packages. Once that has been done, installing the previously listed dependencies becomes simple. In terminal, type `brew search ____` to list packages related to ____, where ____ is your search (for example, GMP, Boost, or MPI). To download new software via Homebrew, type in terminal `brew install ____`
- Matlab: make sure to have it installed, and on the path for the command line.
- Python: use `pip` to install 'dem dependencies

B.3.2 Installation

- Install parallel Bertini from source
  1. Download the source tarball (`.tar.gz`) from bertini.nd.edu
  2. Unpack the tarball (just double click it, if you're into mousing)
  3. If you didn't already, now use Homebrew to install the dependencies for Bertini1
     `brew install autoconf automake libtool mpfr gmp boost mpich`
     ok, Bertini1 doesn't depend on Boost, but Bertini_real does
  4. In the terminal, move to the unpacked tarball for Bertini1
  5. `./configure`
  6. `make -j 4 && make install`
- Install Bertini_real
  1. `which git` if it's emtpy, then `brew install git`
  2. `mkdir code && cd code`
  3. `git clone https://github.com/ofloveandhate/bertini_real`
  4. `cd bertini_real`
  5. `autoreconf -i`
  6. `./configure`
  7. `make -j 4 && make install`

Problems? Raise an issue on GitHub, please.

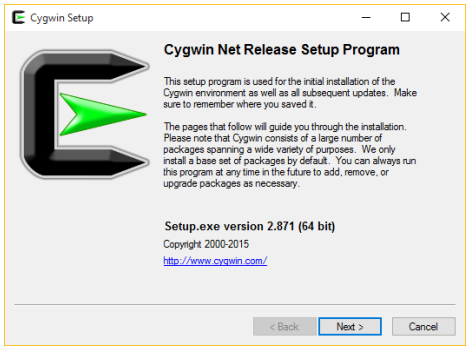
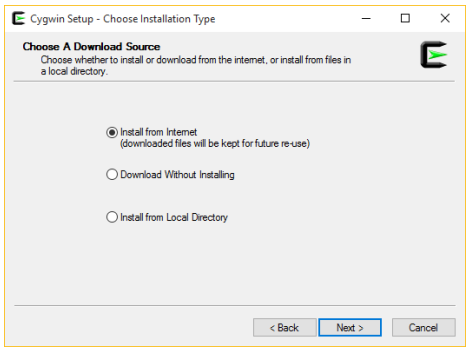
B.4 Instructions for Windows

Unlike Linux or Mac computers, Windows users have additional pre-requisites that they need to install in order to use Bertini and Bertini_real — they need to first install the program Cygwin.[5] Or, maybe MinGW. Alternatively, with Windows 10 and Bash support upcoming, consider using Chocolatey or bash itself. Last checked (fall 2016), Chocolatey did not provide Boost, so that was a bummer.

The other two operating systems that were discussed above were developed with some flavor of *nix, while Windows was not. So, in order to run applications like Bertini that appear to need Linux, we need an intermediary program. Cygwin is a Linux-like environment for Windows.

B.4.1 Install Cygwin

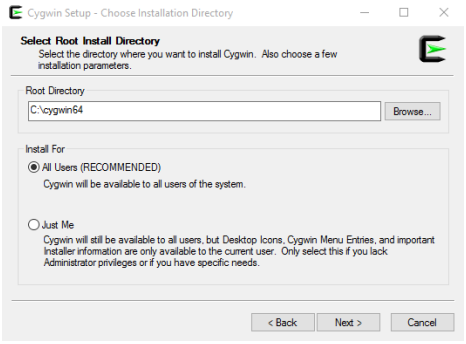
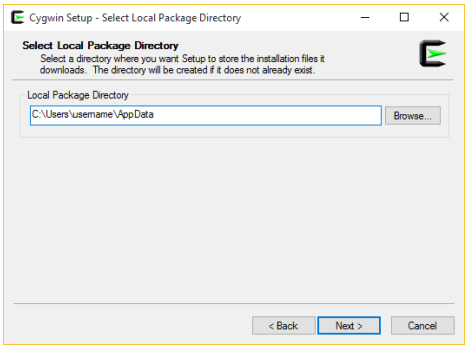
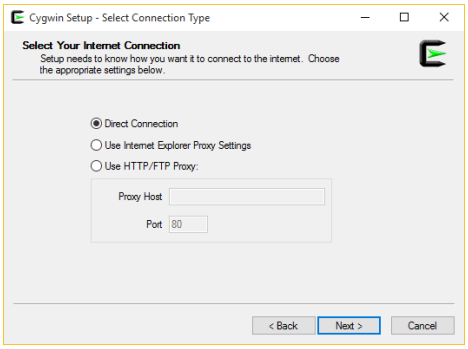
Cygwin can be found at cygwin.com/install.html. Please make sure to choose the version (either 32-bit or 64-bit) that is appropriate for your laptop. After the setup-x86.exe (or setup-x86_64.exe) has downloaded, run it, and follow the instructions.

Instructions	Screen Shot
Click 'next' on the first screen.	
Select the 'Install from Internet' option; click 'next'.	

Continued on next page

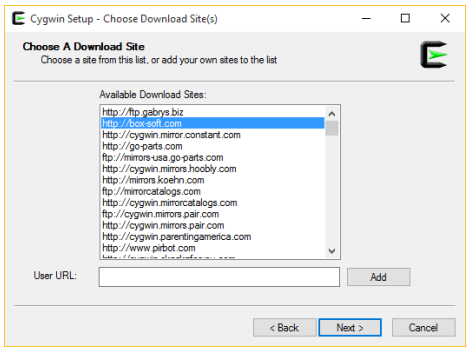
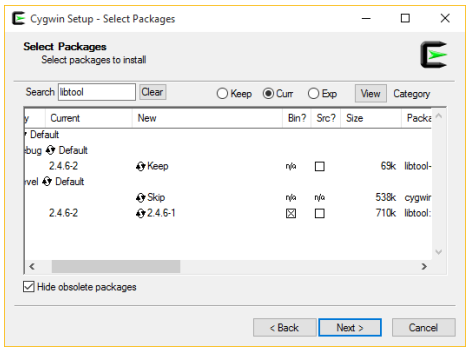
[5] These Windows install instructions prepared by Beth Sudkamp. Thanks Beth!

Table 5: *Continued from previous page*

Instructions	Screen Shot
Enter the preferred installation directory; click 'next'.	
Choose a temporary installation folder; click 'next'.	
Select the 'Direct Connection' option; click 'next'.	

Continued on next page

Table 5: *Continued from previous page*

Instructions	Screen Shot
Choose a download site; click 'next'.	
Select the packages that you will need. , then click 'next'. See below for more instructions	
If during the course of installation, a message pops up and says that certain dependencies are required for the packages, click the 'yes' button. When it has installed everything, select 'finish'.	You're done!

B.4.2 Selecting packages for Cygwin

The list of packages that you will need for Bertini_real can be found below. To find the packages, a user can type the name into the search in the top left of the menu, which will then show the packages containing that name (e.g. 'libtool'). To choose a package click on the text that says 'Skip' until it changes to a version number (e.g. '2.4.6-1').

autoconf	automake
bash	bison
boost (all the C and C++ libraries)	bzip2
X-11	emacs (or nano or some other text editor that you prefer)
flex	vim
mingw-gcc-g++-4.7.3-1	cygutils-X11
gmp	gcc
mpc	mpfr
libzip2	xinit
libtool	openmpi
openssl	openssh
tar	

B.4.3 Initializing Paths

In order to properly run Cygwin, you need to add Cygwin to the PATH variable. In order to do so, follow these steps:

1. Open the Control Panel and select 'System'.

2. Select the 'Advanced System Settings', and then the 'Environment Variables' option.

3. In the window that appears, select the system variable "PATH' and append `; C:\cygwin\bin` to the end of the PATH variable. When you are doing this, you can also append `; C:\path\to\matlab.exe` to the end of the PATH as well.

   After installing MATLAB, please be sure to add `C:\User\username\...\matlab.exe` to your PATH variable.

B.4.4 Organizing Cygwin

Once Cygwin and MATLAB have been added to the PATH variable, you are now ready to open and run Cygwin. For users who are not familiar with Cygwin, a good reference sheet can be found here.

As part of the installation process, Cygwin will automatically configure and install the packages you selected. This is useful, since it saves a lot of time for the user. However, this also allows a Cygwin user to go and be able to automatically use some of these applications, such as 'libtoolize'. Libtoolize, one of the packages that was installed with `setup86x.exe` allows a user to set up a shared library format. In other words, a user doesn't have to call each different library; they are already set up and in the same place.

When I set up Cygwin, I created a new folder located in `\usr\local` that would contain any downloaded files from the Internet that would be used with Cygwin.

When setting up Cygwin, I found that in order to install the dependencies that are needed for Bertini and Bertini_real, they had to be downloaded from the Internet. The following instructions describe how to install these dependencies and finish setting up Cygwin, and were paraphrased from How to Install a Newer Version of GCC.

B.4.5 Installing Dependencies

As stated earlier, the dependencies that need to be downloaded are:

C++ compiler	gcc(Cygwin)	✓
MPI	openmpi(Cygwin)	✓
Boost	boost(Cygwin)	✓
GMP	gmp(Cygwin)	✓
MPFR	here	
GNU Multiple Precision Complex Library (MPC)	here	

Once you have downloaded the programs from the sites, put the zipped files in the folder that you created in
`\usr\local\your_folder`. Then, in Cygwin, enter `your_folder`.

B.4.6 Linking Cygwin Environment Paths

After logging into Cygwin, a user needs to set up their environment paths inside the terminal before they set up the files they downloaded. In order to see how the paths currently are set up, either type the following code into the terminal, or copy it and paste it into the terminal:

```
echo ;\
echo LD_LIBRARY_PATH=${LD_LIBRARY_PATH}; \
echo LIBRARY_PATH=${LIBRARY_PATH}; \
echo CPATH=${CPATH}; \
echo PATH=${PATH}; \
echo    \cite{installnewerGCC}
```

File 15: Adapted from [9]

Some things to keep in mind while setting up the environment variables LD_LIBRARY_PATH, LIBRARY_PATH, and CPATH:

- **LD_LIBRARY_PATH** and **LIBRARY_PATH** should contain /usr/local/lib (**LIBRARY_PATH** shall not be set on Enterprise Linux Enterprise Linux Server release 5.5 (cartage))
- **CPATH** should contain /usr/local/include
- If **PATH** contains `c:/windows/system32` (or `/cygdrive/c/windows/system32`; case-insensitive), it should be after /bin and /usr/bin. Otherwise the scripts will try to run Windows sort.exe instead of the Unix command with the same name.

To change or modify the different variables, you can use the code below (or you can change the variables in the Control Panel, as shown earlier):

```
setenv LD_LIBRARY_PATH /usr/local/lib
setenv LIBRARY_PATH /usr/local/lib
setenv CPATH /usr/local/include
```

File 16: Adapted from [9]

However, if Cygwin shows a message such as `-bash: setenv: command not found`, then you need to use the code below:

```
    export LD_LIBRARY_PATH=/usr/local/lib
# Depending on system, LIBRARY_PATH shall not be set -
#   export LIBRARY_PATH=
    export LIBRARY_PATH=/usr/local/lib
    export CPATH=/usr/local/include
```

File 17: Adapted from [9]

B.4.7 Building and Installing Packages

Now that the environment variables are set up, we can now build and set these packages.

Perform the following build/install steps for the **MPFR** and **MPC** packages *in that order*:

1. cd to your workspace directory (above, e.g., cd `\usr\local\your_folder`)
2. Extract the tarball using tar (e.g., `tar -xf mpfr-3.1.3.tar.bz2`). This will create a sub-folder with the source for the given package cd into that source folder (e.g., `cd mpfr-3.1.3`)
3. Type `libtoolize` into the command line and press enter. This will add the files, once they have been compiled, to the shared library.
4. Generate `configure`, by running the command `autoreconf -i`.
5. Read the README and/or INSTALL file if present
6. Note that for the current version of **mpc (0.9)** there is a change that may need to be made to have the build work successfully. You need to edit the line of "mpc.h"

   `#if defined(__MPC_WITHIN_MPC) && __GMP_LIBGMP_DLL` to `#if defined __GMP_LIBGMP_DLL`
7. run `./configure` (this will check the configuration of your system for the purpose of this package)(you also need specify `--enable-static --disable-shared` when compiling the library)
8. run `make` (this will build the package; `-j` can speed things up here)
9. run `make check` (strongly recommended but optional; this will check that everything is correct)
10. run `make install` (this will install all the relevant files to the relevant directories)
11. run `make clean` (optional; this will erase intermediate files - important if you are re-attempting a broken build!)

B.4.8 Installing Bertini and Bertini_real

Once all of the dependencies have been installed, now Bertini and Bertini_real can be installed. The zip file for Bertini can be found here, while the download site for Bertini_real can be found there

Move these downloads to your terminal, and unzip and install the two programs. To unpack the directory, just run `tar -zxvf FILE_NAME` into the command line while in the folder that the .tar.gz is located. (For Cygwin users, this means going through the same steps as you did for GMP, MPFR, and MPC.) Be sure to install Bertini *before* Bertini_real!!

B.4.9 Setting up MATLAB

After you have set up Bertini and Bertini_real, you probably want to be able to see a 3D rendering of your solutions. In order to do so, go to the GitHub Bertini_real site and download the .zip file. I recommend that a new folder is created and that the zip folder goes inside this 'master folder'. Unfortunately, the user also has to download each of the functions that are on the same level as the zip folder- e.g. `dehomogenize.m`, `find_constant_vars.m`, etc. and save those in the 'master folder' as well. Make sure that this 'master folder' is linked to the folder where the Bertini and Bertini_real solutions will be located. Once all of these functions and the zip folder are downloaded and saved, then you will be able to successfully use MATLAB in conjunction with Bertini and Bertini_real.

Note: if you simply clone the repo, you get the matlab codes, no fuss, no muss. Just use git.

Congratulations, you have now made it through the installation process. There are some additional features that can also be installed if you so desire, as well as a practice run in order to verify that everything installed correctly.

B.5 Testrun – the Cayley Cubic

Now that everything has been installed, we can now do a test run, to make sure that everything is working properly. To test this program, we will try to generate a Cayley Cubic using the above programs, following a number of steps. The first step is to create an input file. Open up a text editor in your terminal and create a file called `input`.

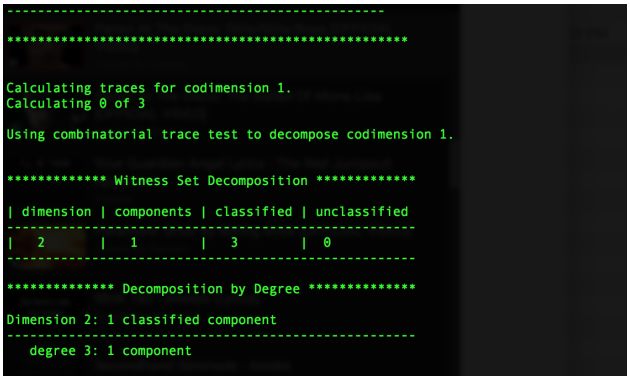
Below is the text for this input file. A key feature to notice is the second line, where the `tracktype` configuration is set to `1`. This configuration setting is necessary for Bertini to run, to run the needed `witness_data` file that Bertini_real uses as input.

```
CONFIG
tracktype:1;

END;
INPUT
variable_group x, y, z;
function f;
f = 4 * (x^2 + y^2 + z^2) + 16*x*y*z - 1;
END;
```

File 18: `input` for the Cayley Cubic

Once the input file is created, we can now run Bertini. Simply navigate in the command line to the directory of the input file and type `bertini` or `bertini input`. You may need to type in the entire pathway to where Bertini is located, if it's not in the same folder, so the command line read `/cygdrive/path/to/BertiniSource_v1.5/bertini-serial.exe input`
Cygwin users: A user may also use `bertini-serial.exe` (or `bertini_parallel.exe`). This will run Bertini, creating the Numerical Irreducible Decomposition needed for Bertini_real. Something like the following should print to the screen:



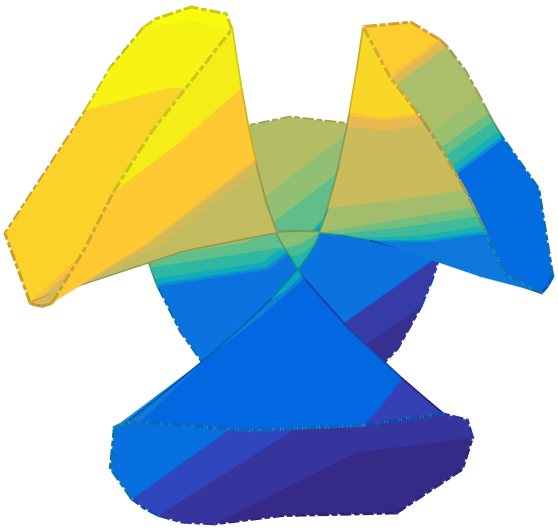
Once Bertini is finished, the output can be verified as satisfactory (or not). Then, Bertini_real can be run by calling `bertini_real` in the command line. Cygwin users, the same rules that applied to Bertini also apply to Bertini_real, so be sure to include that '.exe' at the end! However, if the input file used was named 'input', no file name is needed at the end of the command line. This program should run for roughly 20-30 seconds (ymmv), with the final terminal/shell output appearing below:

```
current midpoint: 458
input_surf_sphere input_surf_sphere

      <<=========   going left
tracking from these point indices:
210 458 209
final top: 8, final bottom:     207
target bottom: 207 current bottom: 207 current top: -12131 final top: 8
tracking to these indices: 207 693 8
u_target=0+1i*0
v_target=5.0000000e-1+1i*-1.6136547e-33
207 bottom_found
8 top_found
found_index of point: 693
added_edge 0, l m r: 207 693 8


          going right   =======>>
tracking from these point indices:
210 458 209
current edge is degenerate, 12==12
decomposed surface has 72 faces
 30.879572s wall, 28.240000s user + 2.440000s system = 30.680000s CPU (99.4%)
```

Finally, MATLAB can be used to visualize the result from the Bertini_real run. Open MATLAB and enter the 'master file', which must be linked to the folder where the Bertini_real solutions are located. This can be done by first making sure that you are currently in the 'master folder', then typing `addpath('C:\cygwin64\path\to\solutions_folder')` into the command window and pressing enter. Then you can call `gather_br_samples` in the command window, which generates a .mat file. Then, call `bertini_real_plotter`. This will create a MATLAB figure, pictured below.



If you've been able to reproduce the above figure, then you've mastered the basics of Bertini_real.

C Output Files

In this section, we describe the formats of the plain-text output files from Bertini_real. We document abstractly, and let the user explore concretely by generating data.

By default, the output from Bertini_real is dumped into a folder named `output_dim_D_comp_C`, where `D` and `C` are the dimension and component numbers. This can be overridden by specifying option `-o` at runtime.

C.1 Regardless of dimension

Some files are produced regardless of object dimension:

- `decomp` – C.1.1
- `input_dim_D_comp_C_deflated` – a plain copy from Bertini_real's input in the containing folder.
- `run_metadata` – C.1.2
- `V.vertex` – C.1.3
- `vertex_types` – C.1.4
- `witness_data` – a plain copy from Bertini's output in the containing folder.
- `witness_set` – C.1.5

Additional important files written NOT into the output folder:

- `Dir_Name` – C.1.6

C.1.1 output/decomp

```
input_filename
number_vars dimension // number_vars includes homvar

FOR // # is dimension
    number_vars_in_projection
    FOR
        proj_coord_real proj_coord_imag
    END FOR
END FOR

number_patches

FOR // # is number_patches
    number_vars_in_patch
    FOR
        patch_coord_real patch_coord_imag
    END FOR
END FOR

sphere_radius_real sphere_radius_imag
num_sphere_vars  // # is number natural variables
FOR // # is number natural variables
    sphere_center_coord_real sphere_center_coord_imag
END FOR

number_crit_fibers
FOR // # is number_crit_fibers
    crit_fiber_coord_real crit_fiber_coord_imag
END FOR
```

File 19: output/decomp

C.1.2 output/run_metadata

```
bertini_real_version
/path/to/containing/folder
timing statistics from Boost.Chrono
```

File 20: output/decomp

C.1.3 output/V.vertex

```
num_vertices num_projections  num_variables  num_filenames //
    number_vars includes homvar

FOR  // num is num_projections
    FOR // num is num coords incl homvar
        proj_coord_real proj_coord_imag
    END FOR
END FOR


FOR  // each filename
    num_chars_in_filename
    filename
END FOR

FOR // each vertex
    num_coords // may include synthetic variables
    FOR
        coord_real coord_imag
    END FOR

    num_projection_values
    FOR
        proj_val_real proj_val_imag
    END FOR

    input_filename_index
    vertex_type  // see file vertex_types
END FOR
```

File 21: `output/V.vertex`

C.1.4 output/vertex_types

```
num_vertex_types

FOR
    VertexType value
END FOR
```

File 22: `output/vertex_types`

Versions of Bertini_real prior to 1.4 used sequential type indices rather than binary values, to vertex types. We converted to binary indices to allow vertices to have multiple types, and represent this compactly. The Matlab visualization code takes advantage of this, and can plot points in each of the type categories they appear in. The purpose of this file is to let one write code which self-adapts to any future vertex types.

C.1.5 output/witness_set

```
num_points dimension component_number

FOR  // each point
    FOR  // each variable
        coord_real coord_imag
    END FOR
END FOR

num_linears num_vars
FOR // each linear
    FOR // each coordinate
        linear_coeff_real linear_coeff_imag
    END FOR
END FOR

num_patches num_vars
FOR // each patch
    FOR // each coordinate
        patch_coeff_real patch_coeff_imag
    END FOR
END FOR
```

File 23: `output/witness_set`

C.1.6 /Dir_Name

```
/path/to/output/folder
MPType  // why?  I don't know, it seemed important early on.
dimension
```

File 24: `output/V.vertex`

C.2 Curve files

There are several files written for every curve decomposed, into the containing folder. For surfaces, each sub-curve is written to its own sub-folder, appropriately named.

- `output/curve.cnums` – C.2.1
- `output/E.edge` – C.2.2
- `output/samp.curvesamp` – C.2.3

C.2.1 `output/curve.cnums`

This file contains the cycle numbers of the paths, tracked from the generic midpoint to the left and right points. Degenerate edges get 0's because there is no path. Otherwise, the numbers are at least 1.

```
number_edges

FOR // each edge
    going_left going_right
END FOR
```

File 25: `output/curve.cnums`

C.2.2 `output/E.edge`

These are 0-based indices into `V.vertex`.

```
number_edges

FOR // each edge
    left mid right
END FOR
```

File 26: `output/E.edge`

C.2.3 `output/samp.curvesamp`

The curve sampling file contains points in order, on edges. The points are referred to by 0-based indices into a new V.vertex file created, preserving the initial one from the decomposition.

```
num_edges

FOR // each edge
     num_samples_on_edge
     FOR // each sample on edge
          sample_index
     END FOR
END FOR
```

File 27: `output/samp.curvesamp`

C.3 Surface files

- `output/F.faces` – C.3.1
- `output/S.surf` – C.3.2
- `output/samp.surfsamp` – C.3.3

C.3.1 `F.faces`

```
number_faces

FOR // each face
    midpoint
    critslice_index
    top_edge_index bottom_edge_index
    system_name_top system_name_bottom

    num_left_edges
    FOR // each left edge
        edge_index
    END FOR

    num_right_edges
    FOR // each right edge
        edge_index
    END FOR
END FOR
```

File 28: `output/F.faces`

C.3.2 `S.surf`

```
number_faces 0 num_midslices num_critslices

number_singular_curves
FOR // each singular curve
    singcurve_multiplicity index
END FOR
```

File 29: `output/S.surf`

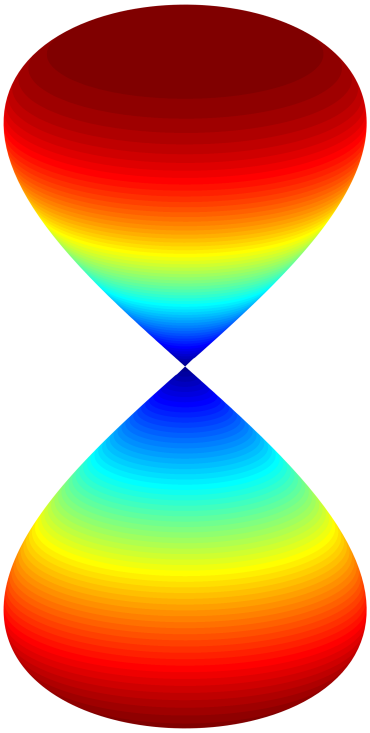
C.3.3 `output/samp.surfsamp`

```
num_faces

FOR // each face
     num_triangles_on_face
     FOR // each triangle
          vert_1 vert_2 vert_3
     END FOR
END FOR
```

File 30: `output/samp.surfsamp`

[type=acronym]



thanks for reading!