PyBertini Documentation

Release 1.0a1

Bertini Team

Jan 22, 2018

Contents

CHAPTER 1

Introductory materials

1.1 Welcome to PyBertini

Bertini is software for numerically solving systems of polynomials. PyBertini is the Python provided for running Bertini.

1.1.1 Mathematical overview

The main algorithm for numerical algebraic geometry implemented in Bertini is homotopy continuation. A homotopy is formed, and the solutions to the start system are continued into the solutions for the target system.

The definitive resource for Bertini 1 is the book *[BHSW13]*. While the way we interact with Bertini changes from version 1 to version 2, particularly when using PyBertini, the algorithms remain fundamentally the same. So do most of the ways to change settings for the path trackers, etc. We believe that embracing the flexibility of Python3 with PyBertini allows for much greater flexibility. It also will relieve the user from the burden of input and output file writing and parsing. Instead, computed results are returned directly to the user.

Consider checking out the *Tutorials*.

1.1.2 Source code

PyBertini is distributed with Bertini2, available at its GitHub repo.

The core is written in template-heavy C++, and is exposed to Python through Boost.Python.

1.1.3 Licenses

Bertini2 and its direct components are available under GPL3, with additional clauses in section 7 to protect the Bertini name. Bertini2 also uses open source softwares, with their own licenses, which may be found in the Bertini2 repository, in the licenses folder.

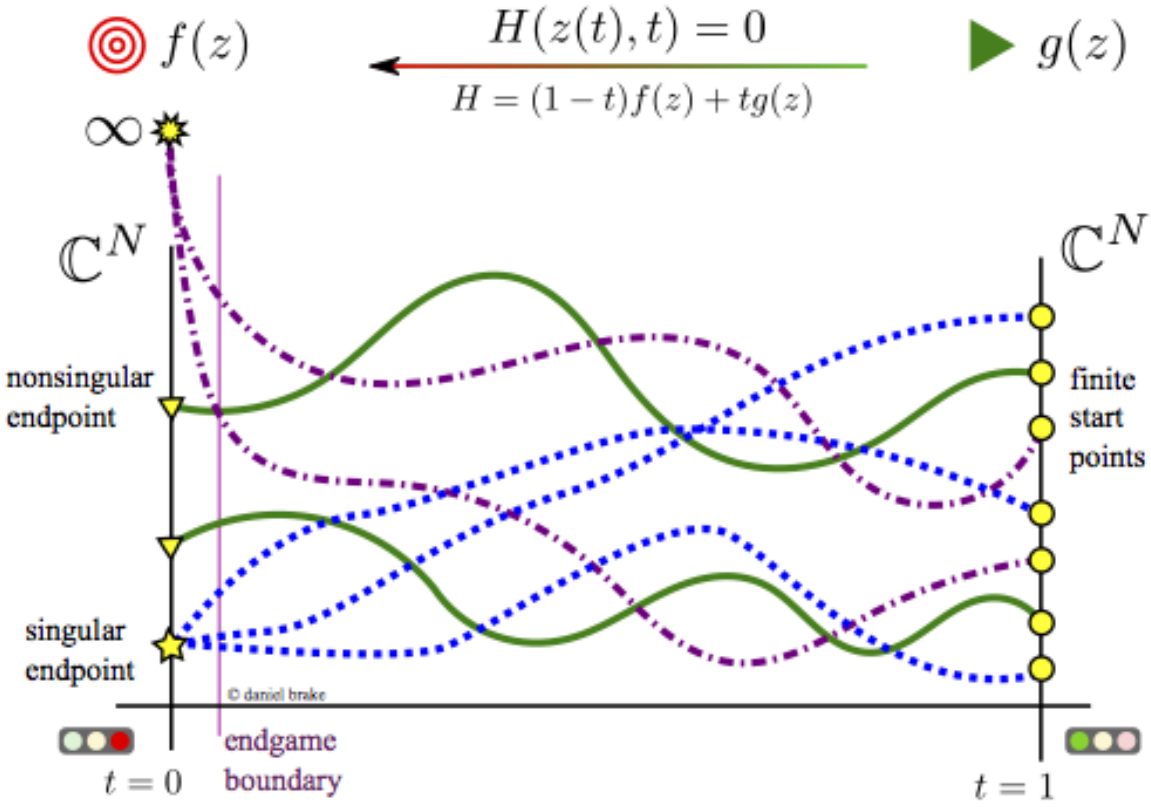

Fig. 1.1: Predictor-corrector methods with optional adaptive precision track paths from 1 to 0, solving f.

1.2 Tutorials

1.2.1 Evaluation of cyclic-n polynomials

Bertini is software for algebraic geometry. This means we work with systems of polynomials, a critical component of which is system and function evaluation.

Bertini2 allows us to set up many kinds of functions, and thus systems, by exploting operator overloading.

Make some symbols

Let's start by making some variables, programmatically[1].

```
import pybertini
import numpy

num_vars = 10
x = [None] * num_vars
for ii in range(num_vars):
    x[ii] = pybertini.Variable('x' + str(ii))
```

Huzzah, we have *num_vars* variables! This was hard to do in Bertini 1's classic style input files. Now we can do it directly!

Write a function to produce the cyclic n polynomials *[DKK03]*.

```
def cyclic(vars):
    n = len(vars)
    f = [None] * len(vars)
    y = []
    for ii in range(2):
        for x in vars:
            y.append(x)

    for ii in range(n):
        f[ii] = numpy.sum( [numpy.prod(y[jj:jj+ii+1]) for jj in range(n)] )

    # the last one is minus one
    f[-1] = f[-1]-1
    return f
```

Now we will make a System, and put the cyclic polynomials into it.

```
sys = pybertini.System()

for f in cyclic(x):
    sys.add_function(f)

print(sys) # long screen output, i know
```

We also need to associate the variables with the system. Unassociated variables are left unknown, and retain their value until elsewhere set.

[1] This is one of the reasons we wrote Bertini2's symbolic C++ core and exposed it to Python.

```
vg = pybertini.VariableGroup()
for var in x:
        vg.append(var)
sys.add_variable_group(vg)
```

Let's simplify this. It will modify elements of the constructed function tree, even those held externally – Bertini uses shared pointers under the hood, so pay attention to where you re-use parts of your functions, because later modification of them without deep cloning will cause ... modification elsewhere, too.

```
pybertini.system.simplify(sys)
```

Now, let's evaluate it at the origin – all zero's (0 is the default value for multiprecision complex numbers in Bertini2). The returned value should be all zero's except the last entry, which should be -1.

```
s = pybertini.multiprec.Vector() # todo allow int in constructor
s.resize(num_vars)
sys.eval(s)
```

Yay, all zeros, except the last one is -1. Huzzah.

Let's change the values of our vector, and re-evaluate.

```
for ii in range(num_vars):
        s[ii] = pybertini.multiprec.complex(ii)
sys.eval(s)
```

There is much more one can do, too! Please write the authors, particularly Dani Brake, for more.

1.2.2 Tracking to nonsingular endpoints

PyBertini works by setting up systems, setting up algorithms to use those systems, and doing something with the output.

Forming a system

Let's make a couple of variables:

```
x = pybertini.function_tree.symbol.Variable("x") #yes, you can make a variable not␣
↪match its name...
y = pybertini.function_tree.symbol.Variable("y")
```

Now, make a few symbolic expressions out of them:

```
f = x**2 + y**2 -1
g = x+y
```

There's no need to "set them equal to 0" – expressions used as functions in a system in Bertini are taken to be equal to zero. If you have an equality that's not zero, move one side to the other.

Let's make an empty system, then build into it:

```
sys = pybertini.System()
sys.add_function(f)
sys.add_function(g)
```

`sys` doesn't know its variables yet, so let's group them into an affine variable group[1], and stuff it into `sys`:

```
grp = pybertini.VariableGroup()
grp.append(x)
grp.append(y)
sys.add_variable_group(grp)
```

Let's check that the degrees of our functions are correct:

```
d = sys.degrees()
assert(d[0]==2)
assert(d[1]==1)
```

What happens if we add a non-polynomial function to our system?

```
sys.add_function(x**-1)
sys.add_function( pybertini.function_tree.sin(x) )
d = sys.degrees()
assert(d[2]==-1) # unsurprising, but actually a coincidence
assert(d[3]==-1) # also -1.  anything non-polynomial is a negative number.  sin has␣
↪no degree
```

Forming a homotopy

A homotopy in Numerical Algebraic Geometry glues together a start system and a target system. Above, we formed a target system, `sys`. Now, let's make a start system `td`, and couple it to `sys`.

The most basic, easiest to form and solve, start system is the Total Degree (TD) start system. It is implemented as a first-class object in Bertini and PyBertini. It takes in a polynomial system as its argument, and self-forms.:

```
del sys #we mal-formed our system above by adding too many functions, and non-
↪polynomial functions to it.
# so, we start over
sys = pybertini.System()
sys.add_variable_group(grp)
sys.add_function(f)
sys.add_function(g)

td = pybertini.TotalDegree(sys)
```

Wonderful, now we have an easy-to-solve system, the structure of which mirrors that of our target system. Every start system comes with a method for generating its start points, by integer index.:

```
# generates the 1th (0-based offsets in python) start point
# at double precision
td.start_point_d(1)

# generate the 1th point at current multiple precision
sp = td.start_point_mp(1)
assert(pybertini.default_precision() == sp[1].precision())
```

Finally, we couple `sys` and `td`:

[1] Affinely-grouped variables live together in the same complex space, $\mathbb{C}^N$. The alternative is projectively-grouped variables, which live in a copy of $\mathbb{P}^N$.

```
t = pybertini.function_tree.symbol.Variable("t")
homotopy = (1-t)*sys + t*td
homotopy.add_path_variable(t)
```

Now, we have the minimum theoretical ingredients for solving a polynomial system using Numerical Algebraic Geometry: a homotopy, a target system, and a start system.

Tracking a single path

There are three basic trackers available in PyBertini:

1. Fixed double precision: `pybertini.tracking.DoublePrecisionTracker`
2. Fixed multiple precision: `pybertini.tracking.MultiplePrecisionTracker`
3. Adaptive precision: `pybertini.tracking.AMPTracker`

Each brings its own advantages and disadvantages. And, each has its ambient numeric type.

Let's use the adaptive one, since adaptivity is generally a good trait to have. `AMPTracker` uses variable-precision vectors and matrices in its ambient work – that is, you feed it multiprecisions, and get back multiprecisions. Internally, it will use double precision when it can, and higher when it has to.

We associate a system with a tracker when we make it. You cannot make a tracker without telling the tracker which system it will be tracking...

```
tr = pybertini.tracking.AMPTracker(homotopy)
tr.set_tolerance(1e-5) # track the path to 5 digits or so

# adjust some stepping settings
stepping = pybertini.tracking.config.SteppingConfig()
stepping.max_step_size = pybertini.multiprec.rational(1,13)

#then, set the config into the tracker.
```

Once we feel comfortable with the configs (of which there are many, see the book or elsewhere in this site, perhaps), we can track a path.

```
result = pybertini.VectorXmp()
tr.track_path(result, pybertini.multiprec.complex(1), pybertini.multiprec.complex(0),␣
↪td.start_point_mp(0))
```

Let's generate a log of what was computed along the way, first making an observer, and then attaching it to the tracker.

```
#make observer

#attach
```

Re-running it, you should find the logfile `bertini#.log`.

Using an endgame to compute singular endpoints

There are two implemented endgames in Bertini:

1. Power series – uses Hermite interpolation across a sequence of geometrically-spaced points (in time) to extrapolate to a target time.
2. Cauchy – uses Cauchy's integral formula

Each is provided in the three precision modes, double, fixed multiple, and adaptive. Since we are using the adaptive tracker in this tutorial, we will of course use the adaptive endgame. I really like the Cauchy endgame, so we're in the land of the `pybertini.endgame.AMPCauchyEG`.

To make an endgame, we need to feed it the tracker that is used to run. There are also config structs to play with, that control the way things are computed.

```
eg = pybertini.endgame.AMPCauchyEG(tr)
```

Since the endgame hasn't been run yet things are empty and default:

```
assert(eg.cycle_number()==0)
assert(eg.final_approximation()==pybertini.VectorXmp())
```

The endgames are used by invoking `run`, feeding it the point we are tracking on, the time we are at, and the time we want to track to.

A complete tracking of paths

```
import pybertini

x = pybertini.function_tree.symbol.Variable("x") #yes, you can make a variable not␣
↪match its name...
y = pybertini.function_tree.symbol.Variable("y")
f = x**2 + y**2 -1
g = x+y

sys = pybertini.System()
sys.add_function(f)
sys.add_function(g)

grp = pybertini.VariableGroup()
grp.append(x)
grp.append(y)
sys.add_variable_group(grp)

td = pybertini.start_system.TotalDegree(sys)

t = pybertini.function_tree.symbol.Variable("t")
homotopy = (1-t)*sys + t*td
homotopy.add_path_variable(t)

tr = pybertini.tracking.AMPTracker(homotopy)

g = pybertini.tracking.observers.amp.GoryDetailLogger()

tr.add_observer(g)
tr.tracking_tolerance(1e-5) # track the path to 5 digits or so
tr.infinite_truncation_tolerance(1e5)
# tr.predictor(pybertini.tracking.Predictor.RK4)
stepping = pybertini.tracking.config.SteppingConfig()
# stepping.max_step_size = pybertini.multiprec.rational(1,13)

results = []

for ii in range(td.num_start_points()):
        results.append(pybertini.multiprec.Vector())
```

```
        tr.track_path(result=results[-1], start_time=pybertini.multiprec.complex(1),␣
↪end_time=pybertini.multiprec.complex(0), start_point=td.start_point_mp(ii))

tr.remove_observer(g)
```

Footnotes

CHAPTER 2

Reference materials

2.1 Detailed

quick nav links:

- jump to *Detailed*
- jump to *Tutorials*

This is a stub page, which merely acts to point you to more specific places in the documentation. Table of contents below .

2.1.1 Highlights

Configurations for algorithms, trackers, endgames, etc

quick nav links:

- jump to *Detailed*
- jump to *Tutorials*

Tracking configs

- `pybertini.tracking.config.SteppingConfig`
- `pybertini.tracking.config.NewtonConfig`
- `pybertini.tracking.config.AMPConfig`
- `pybertini.tracking.config.FixedPrecisionConfig`

Endgame configs

Algorithm configs

2.1.2 Modules

pybertini

quick nav links:

- jump to *Detailed*
- jump to *Tutorials*

Namespaces

- `multiprec`
- `system`
- `function_tree`
- `tracking`
- `endgame`

Convenience

For your convenience, these things have been placed in the root level *pybertini* namespace:

- `System`
- `Variable`
- `VariableGroup`

There's not a whole lot else at this level. Pybertini mostly exists in submodules, to help things be organized.

pybertini.minieigen

quick nav links:

- jump to *Detailed*
- jump to *Tutorials*

Notes

Auto-generated docs

pybertini.doubleprec

quick nav links:

- jump to *Detailed*

- jump to *Tutorials*

Notes

Auto-generated docs

pybertini.multiprec

quick nav links:

- jump to *Detailed*
- jump to *Tutorials*

Notes

Auto-generated docs

pybertini.function_tree

quick nav links:

- jump to *Detailed*
- jump to *Tutorials*

Notes

Auto-generated docs

pybertini.function_tree.symbol

pybertini.function_tree.root

pybertini.system

quick nav links:

- jump to *Detailed*
- jump to *Tutorials*

Notes

Auto-generated docs

pybertini.system.start_system

quick nav links:

- jump to *Detailed*

- jump to *Tutorials*

Notes

Auto-generated docs

pybertini.tracking

quick nav links:

- jump to *Detailed*
- jump to *Tutorials*

Notes

Trackers in Bertini2 are stateful objects, that refer to a system they are tracking, hold their specific settings, and have a notion of current time and space value.

Here are some particular classes and functions to pay attention to:

- `pybertini.tracking.AMPTracker`
- `pybertini.tracking.DoublePrecisionTracker`
- `pybertini.tracking.MultiplePrecisionTracker`

Here are the implemented ODE predictors you can choose from:

- `pybertini.tracking.Predictor`

Calls to `track_path()` return a `pybertini.tracking.SuccessCode`.

Auto-generated docs

pybertini.tracking.config

pybertini.endgame

quick nav links:

- jump to *Detailed*
- jump to *Tutorials*

Notes

Auto-generated docs

pybertini.endgame.config

pybertini.parse

quick nav links:

- jump to *Detailed*
- jump to *Tutorials*

Notes

Auto-generated docs

2.1.3 Things you probably don't need

C++-flavored gory-detail documentation

quick nav links:

- jump to *Detailed*
- jump to *Tutorials*

_pybertini

_pybertini.function_tree

_pybertini.tracking

_pybertini.endgames

2.2 Building PyBertini

This part is unsatisfactory to me. I really wish the package would just detect dependencies, and build itself. However, since there is a C++ library behind it, this is not yet implemented. For now, you have to configure, compile, and install yourself.

Please see the b2 wiki entry for compilation

2.3 Bibliography

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Bibliography

[BHSW13] Daniel J. Bates, Jonathan D. Hauenstein, Andrew J. Sommese, and Charles W. Wampler. *Numerically Solving Polynomial Systems with Bertini*. SIAM, first edition, 2013. ISBN 978-1-611972-69-6.

[DKK03] Yang Dai, Sunyoung Kim, and Masakazu Kojima. Computing all nonsingular solutions of cyclic-n polynomial using polyhedral homotopy continuation methods. *Journal of Computational and Applied Mathematics*, 152(1):83–97, 2003.