phcpy Documentation

Release 1.1.4

Jan Verschelde

Mar 20, 2024

Contents:

CHAPTER 1

Getting Started

This documentation describes a Python package to compute solutions of polynomial systems using PHCpack.

The computation of the mixed volume in phcpy calls MixedVol (ACM TOMS Algorithm 846 of T. Gao, T.Y. Li, M. Wu) as it is integrated in PHCpack. DEMiCs (Dynamic Enumeration of all Mixed Cells, by T. Mizutani, A. Takeda, and M. Kojima) is faster than MixedVol for larger systems with many different supports. A function to compute mixed volumes with DEMiCs is available in phcpy.

For double double and quad double arithmetic, PHCpack incorporates the QD library of Y. Hida, X.S. Li, and D.H. Bailey and code generated by the CAMPARY library. CAMPARY is the CudA Multiple Precision ARithmetic librarY, by Mioara Joldes, Olivier Marty, Jean-Michel Muller, Valentina Popescu and Warwick Tucker.

1.1 what is phcpy?

The main executable phc (polynomial homotopy continuation) defined by the source code in PHCpack is a menu driven and file oriented program. The Python interface defined by phcpy replaces the files with persistent objects allowing the user to work with scripts or in interactive sessions. The computationally intensive tasks such as path tracking and mixed volume computations are executed as compiled code so there will not be a loss of efficiency.

Both phcpy and PHCpack are free and open source software packages: you can redistribute it and/or modify it under the terms of version 3 of the GNU General Public License as published by the Free Software Foundation.

At the poster session EuroSciPy 2013, questions from Max Demenkov led to development of a step-by-step path tracker, a process in which the user can control the pace of the tracker, asking for the next point on a solution path, which is very convenient for plotting. This is one of the features of `phcpy` which makes PHCpack more versatile to the Python programmer.

1.2 installing phcpy

The installation requires the library files `libPHCpack.so` (Linux), `libPHCpack.dll` (windows), and `libPHCpack.dylib` (Mac OS X) to be present in the folder that holds the modules of `phcpy`.

The file `setup.py` defines the instructions to install and is located of the parent folder of `phcpy` modules.

Then the installation happens simply via

```
pip install .
```

executed at the command prompt in the folder where `setup.py` is located. The installation may require superuser priveleges (although virtual environments are recommended); and on older systems with a python2 present, `pip3` may be needed.

To make the library files `libPHCpack`, one can use the alire crate `phcpack`. Alire <https://alire.ada.dev> is the name of the package manager of Ada.

1.3 extending SageMath with phcpy

The SageMath project was one of the motivations for the development of `phcpy`.

To extend SageMath with `phcpy`, locate the python interpreter used by SageMath and then extend that python with `phcpy`.

Importing phcpy apparently changes the configuration of the signal handlers which may lead SageMath to crash when exceptions occur. Thanks to Marc Culler for reporting this problem and for suggesting a work around:

```
sage: import phcpy
sage: from cysignals import init_cysignals
sage: init_cysignals()
sage: pari(1)/pari(0)
```

Without the `init_cysignals()`, the statement `pari(1)/pari(0)` crashes SageMath. With the `init_cysignals()`, the `PariError` exception is handled and the user can continue the SageMath session.

1.4 project history

This section describes some milestones in the development history.

The Python interface to PHCpack got to a first start when Kathy Piret met William Stein at the software for algebraic geometry workshop at the IMA in the Fall of 2006. The first version of this interface is described in the 2008 PhD Thesis of Kathy Piret.

The implementation of the Python bindings depend on the C interface to PHCpack, developed for use with message passing on distributed memory computers.

Version 0.0.1 originated at lecture 40 of MCS 507 in the Fall of 2012, as an illustration of Sphinx. In Spring of 2013, version 0.0.5 was presented at a graduate computational algebraic geometry seminar. Version 0.1.0 was prepared for presentation at EuroSciPy 2013 (August 2013). Improvements using pylint led to version 0.1.1 and the module maps was added in version 0.1.2. Version 0.1.4 added path trackers with a generator so all solutions along a path are returned to the user. Multicore path tracking was added in version 0.1.7.

The paper **Modernizing PHCpack through phcpy** written for the EuroSciPy 2013 proceedings and available at <http://arxiv.org/abs/1310.0056> describes the design of phcpy.

Version 0.2.9 coincides with version 2.4 of PHCpack and gives access to the first version of the GPU accelerated path trackers. Sweep homotopies to explore the parameter space with detection and location of singularities along the solution paths were exported in the module sweepers.py in version 0.3.3 of phcpy. With the addition of a homotopy membership test in verion 0.3.7, the sets.py module provides the key ingredients for a numerical irreducible decomposition. Version 0.5.0 introduced Newton's method on power series. Use cases were added to the documentation in versions 0.5.2, 0.5.3, and 0.5.4. With static linking, the dependencies on the gnat runtime libraries are removed and the Sage python interpreter could be extended with version 0.6.2. Better support of Laurent polynomial systems was added in version 0.6.8. In version 0.6.9, the large module sets.py was divided up, leading to the new modules cascades.py, factor.py, and diagonal.py. Code snippets for jupyter notebook menu extensions were defined in version 0.7.4. Version 0.8.3 gave access to DEMiCs to compute mixed volumes by dynamic enumeration of all mixed cells.

Version 0.9.2 was presented in a talk at the Python devroom at FOSDEM 2019. Another important milestone was the poster presentation at the 18th Python in Science Conference, SciPy 2019, with a paper which appeared in the proceedings (see the references below).

As the original goal of phcpy was on exporting the functionality of PHCpack, its design is functional and phcpy is a collection of modules. Since version 1.0.0, two class definitions were added, one to represent systems of polynomials and another class to represent solutions of polynomial systems.

Prior to release 1.1.3, extension modules were applied, which worked on Linux and Mac OS X, but unfortunately not on Windows. Version 1.1.3 relies directly on the object file `libPHCpack` and applies the `ctypes` to interface more directly with the compiled code. The development of version 1.1.3 happened mainly on Windows, in an Anaconda distribution of Python.

1.5 references

1. T. Gao, T. Y. Li, M. Wu: **Algorithm 846: MixedVol: a software package for mixed-volume computation.** *ACM Transactions on Mathematical Software*, 31(4):555-560, 2005.
2. Y. Hida, X.S. Li, and D.H. Bailey: **Algorithms for quad-double precision floating point arithmetic.** In *15th IEEE Symposium on Computer Arithmetic (Arith-15 2001)*, 11-17 June 2001, Vail, CO, USA, pages 155-162. IEEE Computer Society, 2001. Shortened version of Technical Report LBNL-46996.
3. M. Joldes, J.-M. Muller, V. Popescu, and W. Tucker: **CAMPARY: Cuda Multiple Precision Arithmetic Library and Applications.** In *Mathematical Software - ICMS 2016*, pages 232-240, Springer-Verlag 2016.
4. A. Leykin and J. Verschelde. **Interfacing with the numerical homotopy algorithms in PHCpack.** In N. Takayama and A. Iglesias, editors, *Proceedings of ICMS 2006*, volume 4151 of *Lecture Notes in Computer Science*, pages 354–360. Springer-Verlag, 2006.
5. T. Mizutani and A. Takeda. **DEMiCs: A software package for computing the mixed volume via dynamic enumeration of all mixed cells.** In M. E. Stillman, N. Takayama, and J. Verschelde, editors, *Software for Algebraic Geometry*, volume 148 of The IMA Volumes in Mathematics and its Applications, pages 59-79. Springer-Verlag, 2008.
6. T. Mizutani, A. Takeda, and M. Kojima. **Dynamic enumeration of all mixed cells.** *Discrete Comput. Geom.* 37(3):351-367, 2007.
7. J. Otto, A. Forbes, and J. Verschelde. **Solving Polynomial Systems with phcpy.** In the *Proceedings of the 18th Python in Science Conference (SciPy 2019)*, edited by Chris Calloway, David Lippa, Dillon Niederhut and David Shupe, pages 58-64, 2019.
8. K. Piret: **Computing Critical Points of Polynomial Systems using PHCpack and Python.** PhD Thesis, University of Illinois at Chicago, 2008.
9. A. J. Sommese, J. Verschelde, and C. W. Wampler. **Numerical irreducible decomposition using PHCpack.** In *Algebra, Geometry, and Software Systems*, edited by M. Joswig and N. Takayama, pages 109-130. Springer-Verlag, 2003.

10. J. Verschelde: **Algorithm 795: PHCpack: A general-purpose solver for polynomial systems by homotopy continuation.** *ACM Transactions on Mathematical Software*, 25(2):251–276, 1999.

11. J. Verschelde: **Modernizing PHCpack through phcpy.** In Proceedings of the 6th European Conference on Python in Science (EuroSciPy 2013), edited by Pierre de Buyl and Nelle Varoquaux, pages 71-76, 2014, available at <http://arxiv.org/abs/1310.0056>.

12. J. Verschelde: **Exporting Ada Software to Python and Julia.** *ACM SIGAda Ada Letters* 42(1):76-78, 2022.

13. J. Verschelde and X. Yu: **Polynomial Homotopy Continuation on GPUs.** *ACM Communications in Computer Algebra*, 49(4):130-133, 2015.

1.6 acknowledgments

The PhD thesis of Kathy Piret (cited above) described the development of a first Python interface to PHCpack. The 2008 `phcpy.py` provided access to the blackbox solver, the path trackers, and the mixed volume computation.

In the summer of 2017, Jasmine Otto helped with the setup of jupyterhub and the definition of a SageMath kernel. Code snippets with example uses of `phcpy` in a Jupyter notebook were introduced during that summer. The code snippets, listed in a chapter of this document, provide another good way to explore the capabilities of the software.

This material is based upon work supported by the National Science Foundation under Grants 1115777, 1440534, and 1854513. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

CHAPTER 2

Tutorial

Via some interesting use cases, several features are introduced.

1. If one can solve polynomial systems, then one can solve many problems, as illustrated by the wide variety of application areas.
2. The use cases illustrate the formulation of the polynomial systems via the packages in Python's computational ecosystem, in particular sympy and numpy.
3. The interpretation of the output often happens via plots.

For the plots, Jupyter notebooks are available in the `tests` folder. The development of the use cases happened via short Python scripts.

In the `phcpy` context, solving a problem includes the writing of scripts which formulate the polynomial system that define the problem and the scripts which verify and interpret the solutions.

2.1 A Counter Example to Koushnirenko's Conjecture

This chapter illustrates the counter example of Bertrand Haas, against the Koushnirenko conjecture, executed on one core and on many cores. For the mathematical background, consult:

Bertrand Haas: **A simple counterexample to Kouchnirenko's conjecture.**, *Beitraege zur Algebra und Geometrie/Contributions to Algebra and Geometry*, volume 43, number 1, pages 1 to 8, 2002.

Bertrand constructed a polynomial system with three monomials in every equation and with five positive real roots. Kouchnirenko's conjecture predicted there could only four positive real roots for such a system.

To get the proper wall clock time, we have to be mindful that the Python code calls the compiled functions in the PHCpack library. Therefore, the Python timers will not give accurate timings. Instead, we have to rely on the actual date and time, from the package `datetime` in Python.

```
from datetime import datetime
```

For the plot, the implicit plotting of `sympy` will be used.

```
from sympy import plot_implicit, symbols, Eq
```

From phcpy we import the following functions:

```
from phcpy.dimension import get_core_count
from phcpy.solver import solve
from phcpy.solutions import filter_real
```

2.1.1 solving the system on many cores

The example of Bertrand Haas is defined as

```
H = [ 'x**108 + 1.1*y**54 - 1.1*y;',
      'y**108 + 1.1*x**54 - 1.1*x;' ]
```

According to the theorem of Bézout, we may expect a number of complex solutions equals to the product of the degrees of the polynomials. The square of 108 equals 11664. As the solver computes all complex solutions, executing the following code block takes some time ...

```
print('Solving on one core ...')
wstart = datetime.now()
sols = solve(H)
wstop = datetime.now()
print('  Number of solutions :', len(sols))
print('start time :', wstart)
print(' stop time :', wstop)
print('   elapsed :', wstop - wstart)
```

The output of the above code cell is

```
Solving on one core ...
  Number of solutions : 11664
start time : 2024-01-28 11:57:53.061707
 stop time : 2024-01-28 11:57:59.223344
   elapsed : 0:00:06.161637
```

We can significantly speed up this computation if the computer has many cores.

```
nbcores = get_core_count()
print('Solving on', nbcores, 'cores ...')
wstart = datetime.now()
sols = solve(H, tasks=nbcores)
wstop = datetime.now()
print('  Number of solutions :', len(sols))
print('start time :', wstart)
print(' stop time :', wstop)
print('   elapsed :', wstop - wstart)
```

The output of the above code cell is

```
Solving on 32 cores ...
  Number of solutions : 11664
```

(continues on next page)

(continued from previous page)

```
start time : 2024-01-28 11:58:07.241324
 stop time : 2024-01-28 11:58:08.747874
   elapsed : 0:00:01.506550
```

Compared the *elapsed :* above with the previous one.

2.1.2 extracting the real roots

Rather than eyeballing all 11,664 complex solutions ourselves, we ask to filter the real solutions.

```
realsols = filter_real(sols, tol=1.0e-8, oper='select')
```

The code cell below prints the solutions in *realsols*:

```
for (idx, sol) in enumerate(realsols):
    print('Solution', idx+1, ':')
    print(sol)
```

The output is

```
Solution 1 :
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  0.00000000000000E+00   0.00000000000000E+00
 y :  0.00000000000000E+00   0.00000000000000E+00
== err :  0.000E+00 = rco :  1.000E+00 = res :  0.000E+00 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  9.91489402484465E-01  -2.94004118110142E-49
 y :  9.91489402484465E-01   2.96676882820234E-49
== err :  7.704E-17 = rco :  8.274E-02 = res :  5.773E-15 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  9.99997917489999E-01   9.52445049970774E-46
 y :  9.19904793199125E-01  -1.72639970804817E-42
== err :  2.708E-16 = rco :  1.601E-03 = res :  8.677E-15 =
Solution 4 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  9.99986016402972E-01   1.19248391761152E-37
 y :  9.36266084294562E-01  -2.97164971887874E-34
== err :  2.240E-15 = rco :  2.070E-03 = res :  3.610E-15 =
Solution 5 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
```

(continues on next page)

(continued from previous page)

```
 x :  9.19904793199125E-01  -2.01786978862774E-41
 y :  9.99997917489999E-01   1.02032044433651E-44
== err :  4.384E-16 = rco :  1.601E-03 = res :  8.564E-15 =
Solution 6 :
t :  1.0000000000000E+00   0.0000000000000E+00
m : 1
the solution for t :
 x :  9.36266084294562E-01  -4.40471624223194E-48
 y :  9.99986016402972E-01   3.13214614463929E-51
== err :  9.256E-17 = rco :  2.070E-03 = res :  3.730E-15 =
```

We observe (0, 0) and five additional real positive roots. According to the Koushnirenko conjecture, we would expect no more than four real positive roots.

2.1.3 plotting the curves

In converting the strings in the polynomial system `H` we have to remove the trailing semicolon

```
x, y = symbols('x y')
p0 = eval(H[0][:-1])
p1 = eval(H[1][:-1])
```

Without knowing the precise location of the intersection points, the curves are hard to plot. The code below produces the plot Fig. 2.1.

```
plot0 = plot_implicit(Eq(p0, 0), (x, 0.93, 1.01), (y, 0.93, 1.01),
                      line_color='black', depth=1,
    markers=[{'args': [[0.99148, 0.93626, 0.99998],
                       [0.99148, 0.99998, 0.93626], 'bo']}],
                      axis_center=(0.93, 0.93), show=False)

plot1 = plot_implicit(Eq(p1, 0), (x, 0.93, 1.01), (y, 0.93, 1.01),
                      line_color='red', depth=1,
                      axis_center=(0.93, 0.93), show=False)

plot0.append(plot1[0])
plot0.show()
```

Let us zoom in to another root … Fig. 2.2 is made executing the code below:

```
plot3 = plot_implicit(Eq(p0, 0), (x, 0.9, 1.0025), (y, 0.99, 1.0025),
                      line_color='black', depth=1,
    markers=[{'args': [[0.99148, 0.93626, 0.91990],
                       [0.99148, 0.99998, 0.99999], 'bo']}],
                      axis_center=(0.9, 0.99), show=False)

plot4 = plot_implicit(Eq(p1, 0), (x, 0.9, 1.0025), (y, 0.99, 1.0025),
                      line_color='red', depth=1,
                      axis_center=(0.9, 0.99), show=False)

plot3.append(plot4[0])
plot3.show()
```

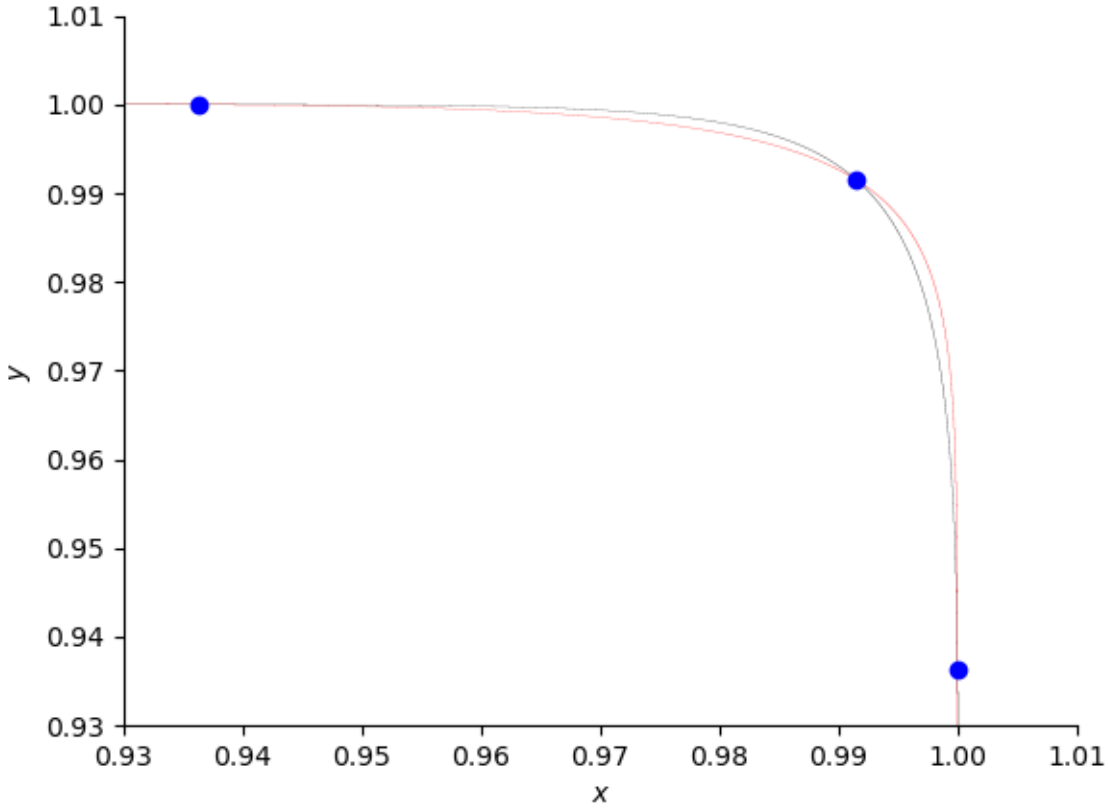


Fig. 2.1: Three positive roots of the counterexample.

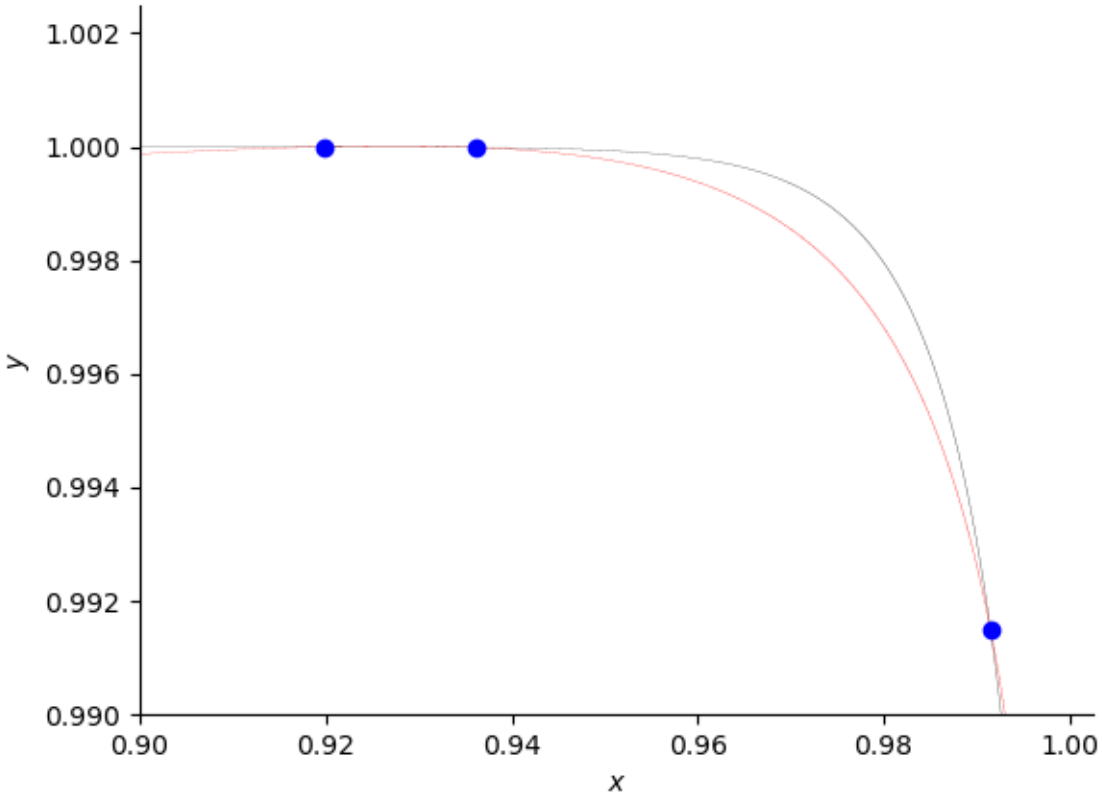


Fig. 2.2: Other positive roots of the counterexample.

The third plot (in Fig. 2.3) is produced by the following code:

```
plot5 = plot_implicit(Eq(p0, 0), (x, 0.9, 1.0005), (y, 0.999, 1.0005),
                      line_color='black', depth=1,
    markers=[{'args': [[0.93626, 0.91990],
                       [0.99998, 0.99999], 'bo']}],
                      axis_center=(0.9, 0.999), show=False)

plot6 = plot_implicit(Eq(p1, 0), (x, 0.9, 1.0005), (y, 0.999, 1.0005),
                      line_color='red', depth=1,
                      axis_center=(0.9, 0.999), show=False)

plot5.append(plot6[0])
plot5.show()
```

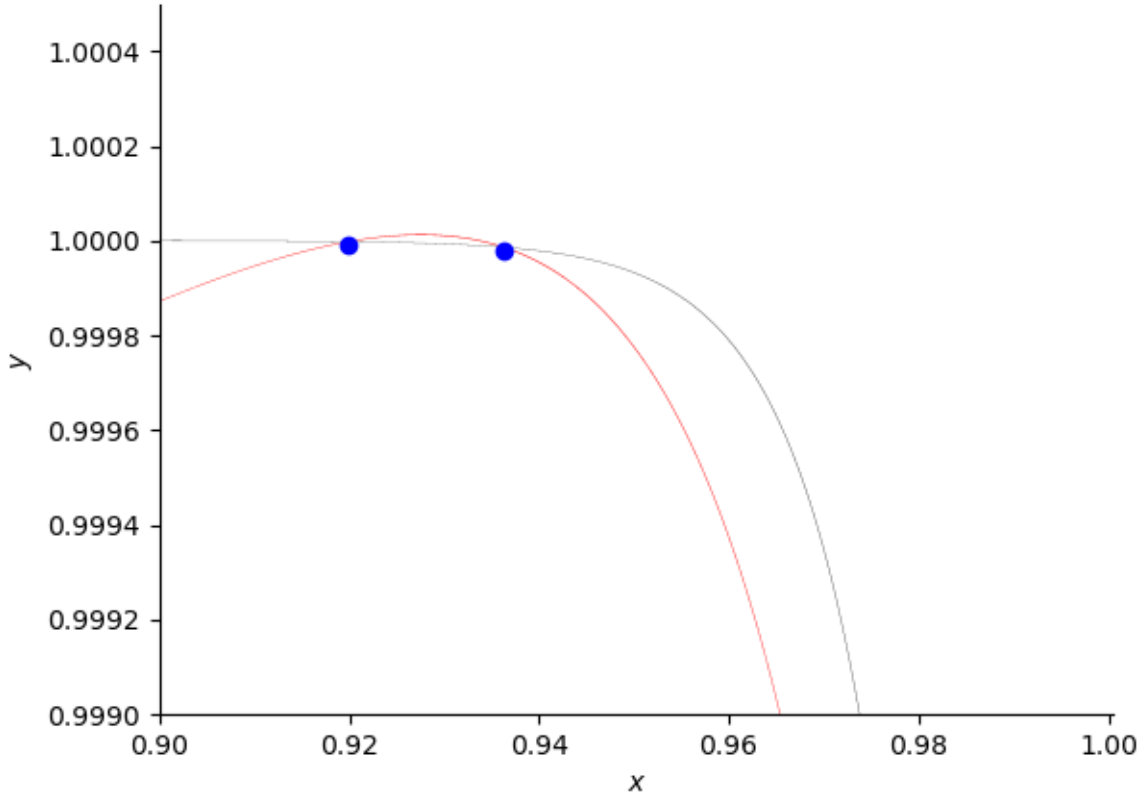


Fig. 2.3: Two close positive roots of the counterexample.

2.2 A 4-Bar Mechanism

The equations to design a 4-bar mechanism are defined with sympy.

The system appears in a paper by A.P. Morgan and C.W. Wampler on **Solving a Planar Four-Bar Design Using Continuation**, published in *the Journal of Mechanical Design*, volume 112, pages 544-550, 1990.

The solutions for a straight-line confirmation are shown with matplotlib. Random numbers will be generated.

```
from math import sqrt
from random import uniform
```

From *sympy* we import the following:

```
from sympy import var
from sympy.matrices import Matrix
```

For the plotting, we import `pyplot` of `matplotlib`.

```
import matplotlib.pyplot as plt
```

And then, last an not least, the blackbox solver of `phcpy` is imported.

```
from phcpy.solver import solve
```

As `phcpy` is an API, the problem is solved via a sequence of functions.

2.2.1 solving a polynomial system

The system of polynomial equations is formulated by the function in the code cell below.

```
def polynomials(d0, d1, d2, d3, d4, a):
    """
    Given in d0, d1, d2, d3, d4 are the coordinates of
    the precision points, given as Matrix objects.
    Also the coordinates of the pivot in a are stored in a Matrix.
    Returns the system of polynomials to design the 4-bar
    mechanism with a coupler passing through the precision points.
    """
    # the four rotation matrices
    c1, s1, c2, s2 = var('c1, s1, c2, s2')
    c3, s3, c4, s4 = var('c3, s3, c4, s4')
    R1 = Matrix([[c1, -s1], [s1, c1]])
    R2 = Matrix([[c2, -s2], [s2, c2]])
    R3 = Matrix([[c3, -s3], [s3, c3]])
    R4 = Matrix([[c4, -s4], [s4, c4]])
    # the first four equations reflecting cos^2(t) + sin^(t) = 1
    p1, p2 = 'c1^2 + s1^2 - 1;', 'c2^2 + s2^2 - 1;'
    p3, p4 = 'c3^2 + s3^2 - 1;', 'c4^2 + s4^2 - 1;'
    # the second four equations on X
    x1, x2 = var('x1, x2')
    X = Matrix([[x1], [x2]])
    c1x = 0.5*(d1.transpose()*d1 - d0.transpose()*d0)
    c2x = 0.5*(d2.transpose()*d2 - d0.transpose()*d0)
```

(continues on next page)

(continued from previous page)

```
    c3x = 0.5*(d3.transpose()*d3 - d0.transpose()*d0)
    c4x = 0.5*(d4.transpose()*d4 - d0.transpose()*d0)
    e1x = (d1.transpose()*R1 - d0.transpose())*X + c1x
    e2x = (d2.transpose()*R2 - d0.transpose())*X + c2x
    e3x = (d3.transpose()*R3 - d0.transpose())*X + c3x
    e4x = (d4.transpose()*R4 - d0.transpose())*X + c4x
    s1, s2 = str(e1x[0]) + ';', str(e2x[0]) + ';'
    s3, s4 = str(e3x[0]) + ';', str(e4x[0]) + ';'
    # the third group of equations on Y
    y1, y2 = var('y1, y2')
    Y = Matrix([[y1], [y2]])
    c1y = c1x - a.transpose()*(d1 - d0)
    c2y = c2x - a.transpose()*(d2 - d0)
    c3y = c3x - a.transpose()*(d3 - d0)
    c4y = c4x - a.transpose()*(d4 - d0)
    e1y = ((d1.transpose() - a.transpose())*R1 \
         - (d0.transpose() - a.transpose()))*Y + c1y
    e2y = ((d2.transpose() - a.transpose())*R2 \
         - (d0.transpose() - a.transpose()))*Y + c2y
    e3y = ((d3.transpose() - a.transpose())*R3 \
         - (d0.transpose() - a.transpose()))*Y + c3y
    e4y = ((d4.transpose() - a.transpose())*R4 \
         - (d0.transpose() - a.transpose()))*Y + c4y
    s5, s6 = str(e1y[0]) + ';', str(e2y[0]) + ';'
    s7, s8 = str(e3y[0]) + ';', str(e4y[0]) + ';'
    return [p1, p2, p3, p4, s1, s2, s3, s4, s5, s6, s7, s8]
```

Let us generate random points and define the polynomial system.

```
pt0 = Matrix(2, 1, lambda i,j: uniform(-1,+1))
pt1 = Matrix(2, 1, lambda i,j: uniform(-1,+1))
pt2 = Matrix(2, 1, lambda i,j: uniform(-1,+1))
pt3 = Matrix(2, 1, lambda i,j: uniform(-1,+1))
pt4 = Matrix(2, 1, lambda i,j: uniform(-1,+1))
# the pivot is a
piv = Matrix([[1], [0]])
equ = polynomials(pt0,pt1,pt2,pt3,pt4,piv)
for pol in equ:
    print(pol)
```

Then the output for random numbers as the parameters is

```
c1^2 + s1^2 - 1;
c2^2 + s2^2 - 1;
c3^2 + s3^2 - 1;
c4^2 + s4^2 - 1;
x1*(-0.275586755195824*c1 + 0.325788266703467*s1 + 0.676839821431551) + x2*(0.
↪325788266703467*c1 + 0.275586755195824*s1 - 0.0938422352018522) - 0.142416227311081;\n
↪",
x1*(0.902513020087508*c2 - 0.151712455719013*s2 + 0.676839821431551) + x2*(-0.
↪151712455719013*c2 - 0.902513020087508*s2 - 0.0938422352018522) + 0.185313955832297;\n
↪",
```

(continues on next page)

(continued from previous page)

```
x1*(-0.44237719048943*c3 - 0.542955104453471*s3 + 0.676839821431551) + x2*(-0.
↪542955104453471*c3 + 0.44237719048943*s3 - 0.0938422352018522) + 0.0117896575671138;\n
↪",
x1*(-0.319438148253377*c4 - 0.397350378412077*s4 + 0.676839821431551) + x2*(-0.
↪397350378412077*c4 + 0.319438148253377*s4 - 0.0938422352018522) - 0.103495227599703;\n
↪",
y1*(-1.27558675519582*c1 + 0.325788266703467*s1 + 1.67683982143155) + y2*(0.
↪325788266703467*c1 + 1.27558675519582*s1 - 0.0938422352018522) - 0.543669293546807;\n",
y1*(-0.0974869799124924*c2 - 0.151712455719013*s2 + 1.67683982143155) + y2*(-0.
↪151712455719013*c2 + 0.0974869799124924*s2 - 0.0938422352018522) - 1.39403888568676;\n
↪",
y1*(-1.44237719048943*c3 - 0.542955104453471*s3 + 1.67683982143155) + y2*(-0.
↪542955104453471*c3 + 1.44237719048943*s3 - 0.0938422352018522) - 0.222672973375007;\n",
y1*(-1.31943814825338*c4 - 0.397350378412077*s4 + 1.67683982143155) + y2*(-0.
↪397350378412077*c4 + 1.31943814825338*s4 - 0.0938422352018522) - 0.460896900777877;\n"
```

The solutions of the polynomial system define a mechanism of which the coupler passes through the five points.

```
sols = solve(equ)
len(sols)
```

The number is 36 which is invariant for this problem. Solving a general problem, for random precision points, shows that the number of solutions is 36.

2.2.2 a straight-line configuration

Let us consider a special problem. Observe the extraction of real solutions in the function below.

```
def straight_line(verbose=True):
    """
    This function solves an instance where the five precision
    points lie on a line.  The coordinates are taken from Problem 7
    of the paper by A.P. Morgan and C.W. Wampler.
    Returns a list of solution dictionaries for the real solutions.
    """
    from phcpy.solutions import strsol2dict, is_real
    pt0 = Matrix([[ 0.50], [ 1.06]])
    pt1 = Matrix([[-0.83], [-0.27]])
    pt2 = Matrix([[-0.34], [ 0.22]])
    pt3 = Matrix([[-0.13], [ 0.43]])
    pt4 = Matrix([[ 0.22], [ 0.78]])
    piv = Matrix([[1], [0]])
    equ = polynomials(pt0,pt1,pt2,pt3,pt4,piv)
    if verbose:
        print('the polynomial system :')
        for pol in equ:
            print(pol)
    sols = solve(equ)
    if verbose:
        print('the solutions :')
        for (idx, sol) in enumerate(sols):
```

(continues on next page)

(continued from previous page)

```
            print('Solution', idx+1, ':')
            print(sol)
        print('computed', len(sols), 'solutions')
    result = []
    for sol in sols:
        if is_real(sol, 1.0e-8):
            soldic = strsol2dict(sol)
            result.append(soldic)
    return result
```

Running the function

```
sols = straight_line()
```

shows

```
the polynomial system :
c1^2 + s1^2 - 1;
c2^2 + s2^2 - 1;
c3^2 + s3^2 - 1;
c4^2 + s4^2 - 1;
x1*(-0.83*c1 - 0.27*s1 - 0.5) + x2*(-0.27*c1 + 0.83*s1 - 1.06) - 0.3059;
x1*(-0.34*c2 + 0.22*s2 - 0.5) + x2*(0.22*c2 + 0.34*s2 - 1.06) - 0.6048;
x1*(-0.13*c3 + 0.43*s3 - 0.5) + x2*(0.43*c3 + 0.13*s3 - 1.06) - 0.5859;
x1*(0.22*c4 + 0.78*s4 - 0.5) + x2*(0.78*c4 - 0.22*s4 - 1.06) - 0.3584;
y1*(-1.83*c1 - 0.27*s1 + 0.5) + y2*(-0.27*c1 + 1.83*s1 - 1.06) + 1.0241;
y1*(-1.34*c2 + 0.22*s2 + 0.5) + y2*(0.22*c2 + 1.34*s2 - 1.06) + 0.2352;
y1*(-1.13*c3 + 0.43*s3 + 0.5) + y2*(0.43*c3 + 1.13*s3 - 1.06) + 0.0440999999999999;
y1*(-0.78*c4 + 0.78*s4 + 0.5) + y2*(0.78*c4 + 0.78*s4 - 1.06) - 0.0784;
```

and then continues with `the solutions :` which is skipped as the output of the function gives the list of real solutions.

```
for (idx, sol) in enumerate(sols):
    (x1v, x2v) = (sol['x1'].real, sol['x2'].real)
    (y1v, y2v) = (sol['y1'].real, sol['y2'].real)
    print('Solution', idx+1, ':')
    print('x = ', x1v, x2v)
    print('y = ', y1v, y2v)
```

The coordinates of the real solutions are shown below.

```
Solution 1 :
x =  -0.0877960434509403 -0.85138690751564
y =  0.235837391307301 -1.41899202703639
Solution 2 :
x =  0.0193359267851516 -0.937757011012446
y =  1.22226669109342 -1.08285087742709
Solution 3 :
x =  -0.595728628822183 -0.617010917712341
y =  0.118171353650905 -1.82939267557673
Solution 4 :
x =  -0.158077261086826 -0.793782551346416
y =  -0.548761782690284 0.278116829722178
```

(continues on next page)

(continued from previous page)

```
Solution 5 :
x =  14.265306631912 -6.51576530896231
y =  -0.621791031677556 -0.0713939584963069
Solution 6 :
x =  -1.79178664902321 1.04613207405924
y =  -1.46486338398045 1.21676347168425
Solution 7 :
x =  0.130643755560844 -0.942516053801942
y =  0.963729735050218 -1.01577587226827
Solution 8 :
x =  -0.358757861563373 -0.537230434093211
y =  0.0870595124133798 1.5543474028655
Solution 9 :
x =  -11.0926159017278 0.450863935272926
y =  -0.396207302280832 -1.04172821286545
Solution 10 :
x =  -0.154697709323186 -0.812626279169727
y =  3.30145715645532 -2.31860323051595
Solution 11 :
x =  -0.0801573081756841 -0.855275240173407
y =  -0.297321862562434 -2.18414388671793
Solution 12 :
x =  0.676178657404253 -0.613650952963839
y =  0.356055523659319 0.310794500797803
Solution 13 :
x =  1.4739209688177 -1.71128474823024
y =  -0.654679846479676 0.028907166911727
Solution 14 :
x =  -0.264640920049152 -0.69691152780256
y =  0.370368746423895 -1.54221173415608
Solution 15 :
x =  -1.0856845753759 -0.352998488913482
y =  0.319028475056347 0.687883260707162
```

2.2.3 a four-bar mechanism

The code in the function below are applied to make the plots.

```
def angle(csa, sna):
    """
    Given in csa and sna are the cosine and sine of an angle a,
    that is: csa = cos(a) and sna = sin(a).
    On return is the angle a, with the proper orientation.
    """
    from math import acos, pi
    agl = acos(csa)
    if sna >= 0:
        return agl
    else:
        dlt = pi - agl
        return pi + dlt
```

```
def angles(soldic):
    """
    Given a solution dictionary, extracts the angles from
    the four cosines and sines of the angles.
    Returns None if the angles are not ordered increasingly.
    Otherwise, returns the sequence of ordered angles.
    """
    from math import acos, asin
    c1v, s1v = soldic['c1'].real, soldic['s1'].real
    c2v, s2v = soldic['c2'].real, soldic['s2'].real
    c3v, s3v = soldic['c3'].real, soldic['s3'].real
    c4v, s4v = soldic['c4'].real, soldic['s4'].real
    ag1 = angle(c1v, s1v)
    ag2 = angle(c2v, s2v)
    ag3 = angle(c3v, s3v)
    ag4 = angle(c4v, s4v)
    ordered = (ag1 > ag2) and (ag2 > ag3) and (ag3 > ag4)
    if ordered:
        print(ag1, ag2, ag3, ag4, 'ordered angles')
        return (ag1, ag2, ag3, ag4)
    return None
```

```
def plotpoints(points):
    """
    Plots the precision points and the pivots.
    """
    xpt = [a for (a, b) in points]
    ypt = [b for (a, b) in points]
    plt.plot(xpt, ypt, 'ro')
    plt.text(xpt[0] - 0.01, ypt[0] + 0.08, \"0\")
    plt.text(xpt[1] - 0.01, ypt[1] + 0.08, \"1\")
    plt.text(xpt[2] - 0.01, ypt[2] + 0.08, \"2\")
    plt.text(xpt[3] - 0.01, ypt[3] + 0.08, \"3\")
    plt.text(xpt[4] - 0.01, ypt[4] + 0.08, \"4\")
    plt.plot([0, 1], [0, 0], 'w^') # pivots marked by white triangles
    plt.axis([-1.0, 1.5, -1.0, 1.5])
```

```
def plotbar(fig, points, idx, x, y):
    """
    Plots a 4-bar with coordinates given in x and y,
    and the five precision points in the list points.
    The index idx is the position with respect to a point in points.
    """
    if idx < 0:
        fig.add_subplot(231, aspect='equal')
    if idx == 0:
        fig.add_subplot(232, aspect='equal')
    elif idx == 1:
        fig.add_subplot(233, aspect='equal')
    elif idx == 2:
        fig.add_subplot(234, aspect='equal')
    elif idx == 3:
```

(continues on next page)

(continued from previous page)

```
        fig.add_subplot(235, aspect='equal')
    elif idx == 4:
        fig.add_subplot(236, aspect='equal')
    plotpoints(points)
    if idx >= 0:
        xpt = [a for (a, b) in points]
        ypt = [b for (a, b) in points]
        (xp0, xp1) = (x[0] + xpt[idx], x[1] + ypt[idx])
        (yp0, yp1) = (y[0] + xpt[idx], y[1] + ypt[idx])
        plt.plot([xp0, yp0], [xp1, yp1], 'go')
        plt.plot([xp0, yp0], [xp1, yp1], 'g')
        plt.text(xp0 - 0.04, xp1 - 0.22, \"x\")
        plt.text(yp0 - 0.04, yp1 - 0.22, \"y\")
        plt.plot([0, xp0], [0, xp1], 'g')
        plt.plot([yp0, 1], [yp1, 0], 'g')
        plt.plot([xp0, xpt[idx]], [xp1, ypt[idx]], 'b')
        plt.plot([yp0, xpt[idx]], [yp1, ypt[idx]], 'b')
```

```
def rotate(x, y, a):
    """
    Applies a planar rotation defined by the angle a
    to the points x and y.
    """
    from sympy.matrices import Matrix
    from math import cos, sin
    rot = Matrix([[cos(a), -sin(a)], [sin(a), cos(a)]])
    xmt = Matrix([[x[0]], [x[1]]])
    ymt = Matrix([[y[0]], [y[1]]])
    rxm = rot*xmt
    rym = rot*ymt
    rox = (rxm[0], rxm[1])
    roy = (rym[0], rym[1])
    return (rox, roy)
```

```
def show4bar():
    """
    Plots a 4-bar design, for the five precision points
    on a straight line, with coordinates taken from Problem 7
    of the Morgan-Wampler paper.
    """
    pt0 = ( 0.50,  1.06)
    pt1 = (-0.83, -0.27)
    pt2 = (-0.34,  0.22)
    pt3 = (-0.13,  0.43)
    pt4 = ( 0.22,  0.78)
    points = [pt0, pt1, pt2, pt3, pt4]
    ags = [1.44734213756, 0.928413708131, 0.751699211109, 0.387116282208]
    x =  (-0.0877960434509, -0.851386907516)
    y =  (0.235837391307, -1.41899202704)
    fig = plt.figure()
    plotbar(fig,points, -1, x, y)
```

(continues on next page)

(continued from previous page)

```
    plotbar(fig,points, 0, x, y)
    rx1, ry1 = rotate(x, y, ags[0])
    plotbar(fig,points, 1, rx1, ry1)
    rx2, ry2 = rotate(x, y, ags[1])
    plotbar(fig,points, 2, rx2, ry2)
    rx3, ry3 = rotate(x, y, ags[2])
    plotbar(fig,points, 3, rx3, ry3)
    rx4, ry4 = rotate(x, y, ags[3])
    plotbar(fig,points, 4, rx4, ry4)
    fig.canvas.draw()
    plt.savefig('fourbarfig1')
```

The mechanism which passes through the precision points is shown in Fig. 2.4 obtained as the output of

```
show4bar()
```

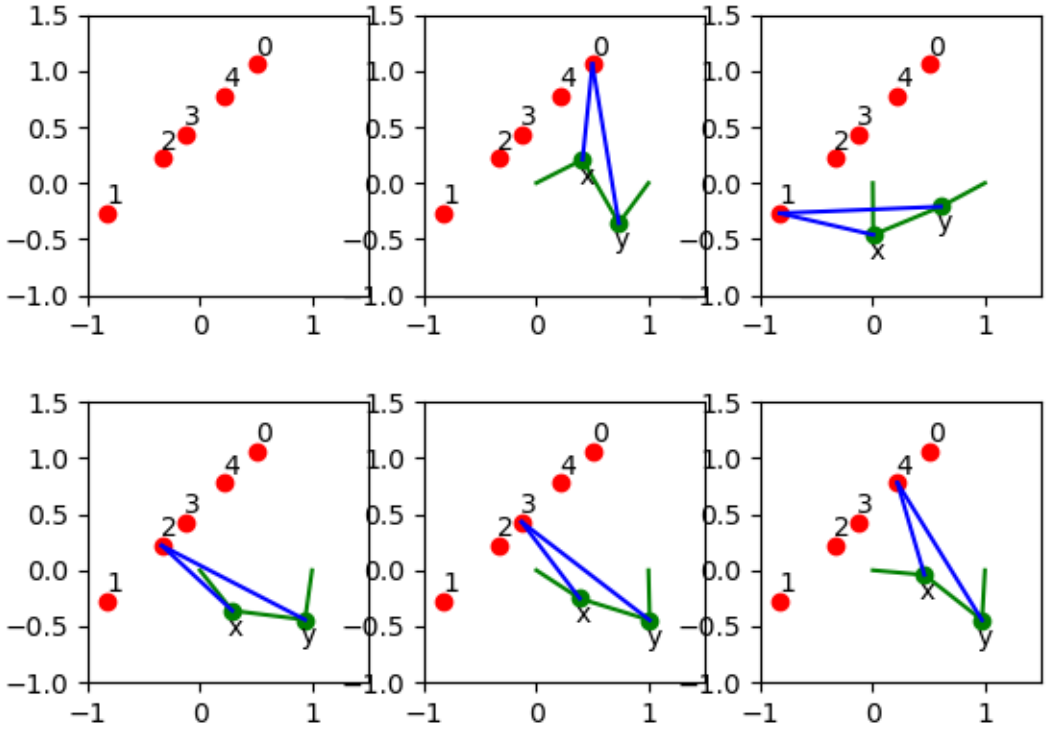


Fig. 2.4: A mechanism passing through precision points.

```
for sol in sols:
    agl = angles(sol)
    if agl != None:
        (x1v, x2v) = (sol['x1'].real, sol['x2'].real)
```

(continues on next page)

(continued from previous page)

```
        (y1v, y2v) = (sol['y1'].real, sol['y2'].real)
        print('x = ', x1v, x2v)
        print('y = ', y1v, y2v)
```

The output is

```
1.4473421375642717 0.9284137081314461 0.75169921110931 0.3871162822082786 ordered angles
x =  -0.0877960434509403 -0.85138690751564
y =  0.235837391307301 -1.41899202703639
2.524711332238134 0.9038272905536054 0.7498546795650226 0.38277375732994035 ordered␣
↪angles
x =  -0.0801573081756841 -0.855275240173407
y =  -0.297321862562434 -2.18414388671793
5.771983513802544 3.9629563185486125 3.442223836627024 0.5242754656511442 ordered angles
x =  0.676178657404253 -0.613650952963839
y =  0.356055523659319 0.310794500797803
```

Observe that one of the lists of ordered angles is used in the `showbar()`.

2.2.4 the coupler curve

The coupler curve is the curve drawn by the coupler point.

```
def plotpoints2(points):
    """
    Plots the precision points and the pivots.
    """
    xpt = [a for (a, b) in points]
    ypt = [b for (a, b) in points]
    plt.plot(xpt, ypt, 'ro')
    plt.text(xpt[0] + 0.01, ypt[0] + 0.06, \"0\")
    plt.text(xpt[1] - 0.03, ypt[1] + 0.06, \"1\")
    plt.text(xpt[2] - 0.01, ypt[2] + 0.06, \"2\")
    plt.text(xpt[3] - 0.01, ypt[3] + 0.06, \"3\")
    plt.text(xpt[4] - 0.01, ypt[4] + 0.06, \"4\")
    plt.plot([0, 1], [0, 0], 'w^') # pivots marked by white triangles
    plt.axis([-1.2, 1.2, -1.0, 1.5])
```

```
def plotbar2(fig, points, idx, x, y):
    """
    Plots a 4-bar with coordinates given in x and y,
    and the five precision points in the list points.
    The index idx is the position with respect to a point in points.
    """
    plotpoints2(points)
    xpt = [a for (a, b) in points]
    ypt = [b for (a, b) in points]
    (xp0, xp1) = (x[0] + xpt[0], x[1] + ypt[0])
    (yp0, yp1) = (y[0] + xpt[0], y[1] + ypt[0])
    if idx >= 0:
        (xp0, xp1) = (x[0] + xpt[idx], x[1] + ypt[idx])
```

(continues on next page)

(continued from previous page)

```
        (yp0, yp1) = (y[0] + xpt[idx], y[1] + ypt[idx])
        plt.plot([xp0, yp0], [xp1, yp1], 'go')
        plt.plot([xp0, yp0], [xp1, yp1], 'g')
        plt.text(xp0 - 0.04, xp1 - 0.12, \"x\")
        plt.text(yp0 - 0.04, yp1 - 0.12, \"y\")
        plt.plot([0, xp0], [0, xp1], 'g')
        plt.plot([yp0, 1], [yp1, 0], 'g')
        plt.plot([xp0, xpt[idx]], [xp1, ypt[idx]], 'b')
        plt.plot([yp0, xpt[idx]], [yp1, ypt[idx]], 'b')
```

```
def lenbar(pt0, x, y):
    """
    In pt0 are the coordinates of the first precision point
    and in x and y the coordinates of the solution design.
    Returns the length of the bar between x and y.
    """
    (xp0, xp1) = (x[0] + pt0[0], x[1] + pt0[1])
    (yp0, yp1) = (y[0] + pt0[0], y[1] + pt0[1])
    result = sqrt((xp0 - yp0)**2 + (xp1 - yp1)**2)
    return result
```

```
def coupler(x, y, xr, yr):
    """
    In x and y are the coordinates of the solution design.
    In xr and yr are the distances to the coupler point.
    Computes the intersection between two circles, centered
    at x and y, with respective radii in xr and yr.
    """
    A = -2*x[0] + 2*y[0]
    B = -2*x[1] + 2*y[1]
    C = x[0]**2 + x[1]**2 - xr**2 - y[0]**2 - y[1]**2 + yr**2
    fail = True
    if A + 1.0 != 1.0: # eliminate z1
        (alpha, beta) = (-C/A, -B/A)
        a = beta**2 + 1
        b = 2*alpha*beta - 2*x[1] - 2*x[0]*beta
        c = alpha**2 + x[0]**2 + x[1]**2 - xr**2 - 2*x[0]*alpha
        if b**2 - 4*a*c >= 0:
            fail = False
            disc = sqrt(b**2 - 4*a*c)
            z2 = (-b + disc)/(2*a)
            z1 = alpha + beta*z2
    if fail:
        (alpha, beta) = (-C/B, -A/B)
        a = beta**2 + 1
        b = 2*alpha*beta - 2*y[1] - 2*y[0]*beta
        c = alpha**2 + y[0]**2 + y[1]**2 - yr**2 - 2*y[0]*alpha
        disc = sqrt(b**2 - 4*a*c)
        z1 = (-b + disc)/(2*a)
        z2 = alpha + beta*z1
        dxz = sqrt((x[0]-z1)**2 + (x[1]-z2)**2)
    return (z1, z2)
```

```
def xcrank(pt0, x):
    """
    In pt0 are the coordinates of the first precision point
    and in x the coordinates of the solution design.
    This function computes the length of the crank
    and its initial angle with respect to the first point.
    """
    from math import atan
    (xp0, xp1) = (x[0] + pt0[0], x[1] + pt0[1])
    crklen = sqrt(xp0**2 + xp1**2)
    crkagl = atan(xp1/xp0)
    return (crklen, crkagl)
```

```
def ycrank(pt0, y):
    """
    In pt0 are the coordinates of the first precision point
    and in y the coordinates of the solution design.
    This function computes the length of the crank
    and its initial angle with respect to the first point.
    """
    from math import cos, sin, acos, pi
    (yp0, yp1) = (y[0] + pt0[0], y[1] + pt0[1])
    crklen = sqrt((yp0 - 1)**2 + yp1**2)
    crkagl = acos((yp0-1)/crklen)
    if yp1 < 0:
        dlt = pi - crkagl
        crkagl = pi + dlt
    cx = 1 + crklen*cos(crkagl)
    cy = crklen*sin(crkagl)
    return (crklen, crkagl)
```

```
def xpos(y1, y2, dxy, rad):
    """
    Given in y1 and y2 are the coordinates of the point y,
    in dxy is the distance between the points x and y,
    and rad is the distance between x and (1, 0).
    The coordinates of the point x are returned in a tuple.
    """
    A = -2*y1  # coefficient with y1
    B = -2*y2  # coefficient with y2
    C = y1**2 + y2**2 - dxy**2 + rad**2 # constant
    fail = True
    if abs(y2) < 1.0e-8:
        x1 = -C/A
        x2sqr = rad**2 - x1**2
        x2 = sqrt(x2sqr)
        fail = False
    else: # eliminate x2
        (alpha, beta) = (-C/B, -A/B)
        (a, b, c) = (1+beta**2, 2*alpha*beta, alpha**2 - rad**2)
        b4ac = b**2 - 4*a*c
        disc = sqrt(b4ac)
```

(continues on next page)

(continued from previous page)

```
        x1m = (-b - disc)/(2*a)
        x2m = alpha + beta*x1m
        x1p = (-b + disc)/(2*a)
        x2p = alpha + beta*x1p
    return ((x1m, x2m), (x1p, x2p))
```

```
def plotcrank(crk, agl, dxy, rad, xrd, yrd):
    """
    Plots several positions of the crank.  On input are:
    crk : length of the crank from the point y to (1, 0),
    agl : start angle,
    rad : length of the crank from (0, 0) to the point x,
    xrd : length from the point x to the coupler point,
    yrd : length from the point y to the coupler point.
    """
    from math import sin, cos, pi
    (xzm, yzm) = ([], [])
    (xzp, yzp) = ([], [])
    nbr = 205
    inc = (pi+0.11763)/nbr
    b = agl - 2.558 # 125
    for k in range(nbr):
        (y1, y2) = (1 + crk*cos(b), crk*sin(b))
        (xm, xp) = xpos(y1, y2, dxy, rad)
        (x1m, x2m) = xm
        (x1p, x2p) = xp
        (z1m, z2m) = coupler([x1m, x2m], [y1, y2], xrd, yrd)
        (z1p, z2p) = coupler([x1p, x2p], [y1, y2], xrd, yrd)
        xzm.append(z1m)
        yzm.append(z2m)
        xzp.append(z1p)
        yzp.append(z2p)
        if k < 0: # selective plot
            plt.plot([0, x1m], [0, x2m], 'g')
            plt.plot([x1m, y1], [x2m, y2], 'g')
            plt.plot([y1, 1], [y2, 0], 'g')
            dyp = sqrt((y1-1)**2 + y2**2)
            dyx = sqrt((x1m-y1)**2 + (x2m-y2)**2)
            print('dxy =', dxy, 'dyp =', dyp)
        if k < 0:
            print('y2 =', y2)\
            plt.plot([x1m, z1m], [x2m, z2m], 'b')
            plt.plot([y1, z1m], [y2, z2m], 'b')
            plt.plot([x1p, z1p], [x2p, z2p], 'b')
            plt.plot([y1, z1p], [y2, z2p], 'b')
        b = b + inc
    plt.plot(xzp[:1]+xzm[:102]+xzp[102:], \
             yzp[:1]+yzm[:102]+yzp[102:], 'r')
    plt.plot(xzp[:102]+xzm[102:], yzp[:102]+yzm[102:], 'r')
```

```
def plotcoupler():
```

(continues on next page)

(continued from previous page)

```
    """
    Plots the coupler curve for a straight line 4-bar mechanism.
    """
    pt0 = ( 0.50,  1.06)
    pt1 = (-0.83, -0.27)
    pt2 = (-0.34,  0.22)
    pt3 = (-0.13,  0.43)
    pt4 = ( 0.22,  0.78)
    points = [pt0, pt1, pt2, pt3, pt4]
    ags = [1.44734213756, 0.928413708131, 0.751699211109, 0.387116282208]
    x = (-0.0877960434509, -0.851386907516)
    y = (0.235837391307, -1.41899202704)
    (xcrk, xagl) = xcrank(pt0, x)
    (ycrk, yagl) = ycrank(pt0, y)
    dxy = lenbar(pt0, x, y)
    fig = plt.figure()
    fig.add_subplot(111, aspect='equal')
    xrd = sqrt(x[0]**2 + x[1]**2) # distance from x to pt0
    yrd = sqrt(y[0]**2 + y[1]**2) # distance from y to pt0
    plotcrank(ycrk, yagl, dxy, xcrk, xrd, yrd)
    plotbar2(fig, points, 0, x, y)
    fig.canvas.draw()
    plt.savefig('fourbarfig2')
```

Running the function

```
plotcoupler()
```

produces the plot in Fig. 2.5.

2.3 Two Lines Meeting Four Given Lines

Given four lines in general position, there are two lines which meet all four given lines. With Pieri homotopies we can solve this Schubert problem. For the verification of the intersection conditions, `numpy` is used. The plots are made with `matplotlib`.

We use random numbers and for reproducible plots, fix the seed.

```
from random import seed
```

From `numpy` we import the following.

```
from numpy import zeros, array, concatenate, matrix
from numpy.linalg import det, solve
```

The plots are in 3-space.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

From *phcpy* we import the following functions:

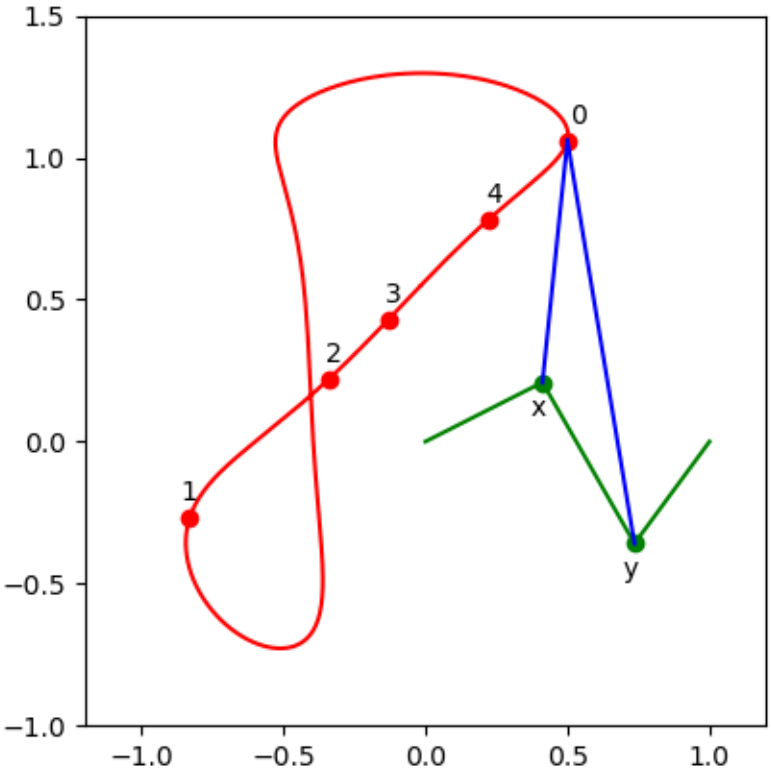


Fig. 2.5: The coupler curve of a 4-bar mechanism.

```
"from phcpy.solutions import coordinates
"from phcpy.schubert import random_complex_matrix
"from phcpy.schubert import pieri_root_count, run_pieri_homotopies
"from phcpy.schubert import real_osculating_planes
"from phcpy.schubert import make_pieri_system
"from phcpy.trackers import double_track as track
```

2.3.1 solving a general instance

A random instance of the four given lines will lead to two solution lines. The formal root count run as

```
(mdim, pdim, deg) = (2, 2, 0)
pcnt = pieri_root_count(mdim, pdim, deg, False)
pcnt
```

and outputs 2.

To setup the problem, some auxiliary functions are first defined.

```
def indices(name):
    """
    For the string name in the format xij
    return (i, j) as two integer indices.
    """
    return (int(name[1]), int(name[2]))
```

```
def solution_plane(rows, cols, sol):
    """
    Returns a sympy matrix with as many rows
    as the value of rows and with as many columns
    as the value of columns, using the string
    represention of a solution in sol.
    """
    result = zeros((rows, cols), dtype=complex)
    for k in range(cols):
        result[k][k] = 1
    (vars, vals) = coordinates(sol)
    for (name, value) in zip(vars, vals):
        i, j = indices(name)
        result[i-1][j-1] = value
    return result
```

```
def verify_determinants(inps, sols, verbose=True):
    """
    Verifies the intersection conditions with determinants,
    concatenating the planes in inps with those in the sols.
    Both inps and sols are lists of numpy arrays.
    Returns the sum of the absolute values of all determinants.
    If verbose, then for all solutions in sols, the computed
    determinants are printed to screen.
    """
    checksum = 0
```

(continues on next page)

(continued from previous page)

```
    for sol in sols:
        if verbose:
            print('checking solution\\n', sol)
        for plane in inps:
            cat = concatenate([plane, sol], axis=-1)
            mat = matrix(cat)
            dcm = det(mat)
            if verbose:
                print('the determinant :', dcm)
            checksum = checksum + abs(dcm)
    return checksum
```

```
def solve_general(mdim, pdim, qdeg):
    """
    Solves a general instance of Pieri problem, computing the
    p-plane producing curves of degree qdeg which meet a number
    of general m-planes at general interpolation points,
    where p = pdim and m = mdim on input.
    For the problem of computing the two lines which meet
    four general lines, mdim = 2, pdim = 2, and qdeg = 0.
    Returns a tuple with four lists.
    The first two lists contain matrices with the input planes
    and the solution planes respectively.
    The third list is the list of polynomials solved
    and the last list is the solution list.
    """
    dim = mdim*pdim + qdeg*(mdim+pdim)
    ranplanes = [random_complex_matrix(mdim+pdim, mdim) for _ in range(0, dim)]
    (pols, sols) = run_pieri_homotopies(mdim, pdim, qdeg, ranplanes)
    inplanes = [array(plane) for plane in ranplanes]
    outplanes = [solution_plane(mdim+pdim, pdim, sol) for sol in sols]
    return (inplanes, outplanes, pols, sols)
```

```
(inp, otp, pols, sols) = solve_general(mdim, pdim, deg)
```

The four input lines are represented as matrices.

```
for plane in inp:
    print(plane)
```

shows

```
[[ 0.98771734-0.15625123j  0.52929265-0.84843933j]
 [ 0.0108879 -0.99994073j  0.43271012+0.90153311j]
 [ 0.670366  +0.74203061j  0.84995049-0.52686257j]
 [-0.99870177+0.05093886j  0.55311134-0.83310735j]]
[[ 0.1176291 +0.9930576j   0.73982601-0.67279824j]
 [-0.4096813 -0.91222872j  0.98222659+0.18769903j]
 [ 0.49367521+0.86964635j -0.00101345-0.99999949j]
 [ 0.99603164-0.0889999j   0.37233497-0.92809841j]]
[[-0.86632581+0.49947932j  0.99954174-0.03027052j]
 [ 0.26897023+0.96314849j  0.29943145+0.95411781j]
```

(continues on next page)

(continued from previous page)

```
 [ 0.77919846-0.62677728j  0.52235751-0.85272659j]
 [ 0.4481898 +0.89393842j  0.97691942+0.21360816j]]
[[ 0.40705515-0.91340358j -0.66900116+0.74326136j]
 [-0.11164153+0.99374854j -0.51718407-0.8558742j ]
 [-0.01384859+0.9999041j  -0.38779064+0.92174748j]
 [ 0.32407475-0.94603148j  0.87995025-0.47506584j]]
```

```
print('The solution planes :')
for plane in otp:
    print(plane)
```

has as output

```
The solution planes :
[[ 1.         +0.j          0.         +0.j        ]
 [-0.64379718+0.67758706j  1.         +0.j        ]
 [ 0.69735824-0.15805905j -1.46030164-0.68747669j]
 [ 0.         +0.j         -1.74595349+0.00175246j]]
[[ 1.         +0.j          0.         +0.j        ]
 [ 1.4746012 +0.78327696j  1.         +0.j        ]
 [ 1.20071164-2.11957742j  0.91569812-1.31875637j]
 [ 0.         +0.j         -1.04202682+0.09584754j]]
```

To check the solutions, we use `numpy` as follows:

```
check = verify_determinants(inp, otp)
print('Sum of absolute values of determinants :', check)
```

The output of the check is

```
checking solution
[[ 1.         +0.j          0.         +0.j        ]
 [-0.64379718+0.67758706j  1.         +0.j        ]
 [ 0.69735824-0.15805905j -1.46030164-0.68747669j]
 [ 0.         +0.j         -1.74595349+0.00175246j]]
the determinant : (2.9667224835639593e-15+1.3550262739027277e-15j)
the determinant : (4.195866422887001e-15-1.4293281742484199e-15j)
the determinant : (-1.8017495082844853e-15-1.5770416093056093e-15j)
the determinant : (-2.0927676352675787e-16+1.091663409852285e-15j)
checking solution
[[ 1.         +0.j          0.         +0.j        ]
 [ 1.4746012 +0.78327696j  1.         +0.j        ]
 [ 1.20071164-2.11957742j  0.91569812-1.31875637j]
 [ 0.         +0.j         -1.04202682+0.09584754j]]
the determinant : (1.0002339027616943e-14-3.132413944024583e-14j)
the determinant : (2.8791053191246284e-14-3.6564204184655514e-15j)
the determinant : (-3.605052372912635e-14+5.874582883240587e-15j)
the determinant : (-2.6498852748806624e-14-2.7706915851697867e-15j)
Sum of absolute values of determinants : 1.362741358344356e-13
```

Observe that all determines evaluate to numbers close to machine precision.

2.3.2 four real lines

We can generate inputs for which all solutions are real.

```
def solve_real(mdim, pdim, start, sols):
    """
    Solves a real instance of Pieri problem, for input planes
    of dimension mdim osculating a rational normal curve.
    On return are the planes of dimension pdim.
    """
    oscplanes = real_osculating_planes(mdim, pdim, 0)
    target = make_pieri_system(mdim, pdim, 0, oscplanes, is_real=True)
    gamma, rtsols = track(target, start, sols)
    print('The solutions to the real problem :')
    for (idx, sol) in enumerate(rtsols):
        print('Solution', idx+1, ':')
        print(sol)
    inplanes = [array(plane) for plane in oscplanes]
    outplanes = [solution_plane(mdim+pdim, pdim, sol) for sol in rtsols]
    return (inplanes, outplanes, target, rtsols)
```

For visualization, the seed of the random number generators is set fixed.

```
seed(400)
```

The output of

```
(oscp, otp2, pols2, sols2) = solve_real(mdim, pdim, pols, sols)
```

is

```
The solutions to the real problem :
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x21 : -2.84638025557899E-02   1.31371731030452E-46
 x32 : -1.19348750548289E-01  -2.62743462060903E-46
 x42 : -4.99706612461873E+00   2.38220738935219E-44
 x31 : -1.06771882518925E+00   3.15292154473084E-45
== err :  5.410E-15 = rco :  5.611E-03 = res :  5.551E-16 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x21 : -5.52734869685360E-02   5.47382212626882E-48
 x32 : -1.19348750548290E-01   4.37905770101505E-47
 x42 : -2.57330433323918E+00   3.83167548838817E-47
 x31 : -6.60558824288729E-01   1.91583774419409E-47,
== err :  6.174E-16 = rco :  1.324E-02 = res :  3.747E-16 =
```

```
print('The input planes :')
for plane in oscp:
    print(plane)
```

```
The input planes :
[[-0.63223829 -0.07958136]
 [ 0.24317589 -0.42625018]
 [ 0.44517428  0.75891681]
 [-0.58562795  0.48582185]]
[[-0.63156273  0.07848797]
 [ 0.31098671 -0.49445305]
 [ 0.32529788  0.85734178]
 [-0.63134544  0.11966993]]
[[-0.66765465 -0.21150281]
 [-0.41782225 -0.46153796]
 [ 0.14470336 -0.77816698]
 [ 0.59893469 -0.36973696]]
[[-0.69033039  0.11246161]
 [-0.09104114 -0.32159814]
 [ 0.66631728 -0.28602371]
 [ 0.26678969  0.89561011]]
```

```
print('The solution planes :')
for plane in otp2:
    print(plane)
```

```
The solution planes :
[[ 1.         +0.00000000e+00j  0.         +0.00000000e+00j]
 [-0.0284638 +1.31371731e-46j  1.         +0.00000000e+00j]
 [-1.06771883+3.15292154e-45j -0.11934875-2.62743462e-46j]
 [ 0.         +0.00000000e+00j -4.99706612+2.38220739e-44j]]
[[ 1.         +0.00000000e+00j  0.         +0.00000000e+00j]
 [-0.05527349+5.47382213e-48j  1.         +0.00000000e+00j]
 [-0.66055882+1.91583774e-47j -0.11934875+4.37905770e-47j]
 [ 0.         +0.00000000e+00j -2.57330433+3.83167549e-47j]]
```

Let us verify the real solution planes as well:

```
check = verify_determinants(oscp, otp2)
print('Sum of absolute values of determinants :', check)
```

Observe the output of the verification:

```
checking solution
[[ 1.         +0.00000000e+00j  0.         +0.00000000e+00j]
 [-0.0284638 +1.31371731e-46j  1.         +0.00000000e+00j]
 [-1.06771883+3.15292154e-45j -0.11934875-2.62743462e-46j]
 [ 0.         +0.00000000e+00j -4.99706612+2.38220739e-44j]]
the determinant : (2.7334976213462325e-15-2.490244814186718e-45j)
the determinant : (6.194410394095717e-15-2.4210378066256254e-45j)
the determinant : (6.1256567148522274e-15-4.974841059851325e-47j)
the determinant : (-1.7538510134158814e-15-1.6274706457865366e-45j)
checking solution
[[ 1.         +0.00000000e+00j  0.         +0.00000000e+00j]
 [-0.05527349+5.47382213e-48j  1.         +0.00000000e+00j]
 [-0.66055882+1.91583774e-47j -0.11934875+4.37905770e-47j]
 [ 0.         +0.00000000e+00j -2.57330433+3.83167549e-47j]]
```

(continues on next page)

(continued from previous page)

```
the determinant : (-6.163408511151722e-16-7.868415222942327e-49j)
the determinant : (3.1253636658440115e-16-1.6674497687062525e-48j)
the determinant : (-1.4639612348256832e-16-1.9640570003033534e-47j)
the determinant : (-1.3795665037633665e-15+8.091364203252659e-48j)
Sum of absolute values of determinants : 1.926225558865557e-14
```

Observe the size of the values of the determinants.

2.3.3 visualization

The code in the functions below help visualizing the problem.

```
def input_generators(plane):
    """
    Given in plane is a numpy matrix, with in its columns
    the coordinates of the points which span a line, in 4-space.
    The first coordinate must not be zero.
    Returns the affine representation of the line,
    after dividing each generator by its first coordinate.
    """
    pone = list(plane[:,0])
    ptwo = list(plane[:,1])
    aone = [x/pone[0] for x in pone]
    atwo = [x/ptwo[0] for x in ptwo]
    return (aone[1:], atwo[1:])
```

```
def output_generators(plane):
    """
    Given in plane is a numpy matrix, with in its columns
    the coordinates of the points which span a line, in 4-space.
    The solution planes follow the localization pattern
    1, *, *, 0 for the first point and 0, 1, *, * for
    the second point, which means that the second point
    in standard projective coordinates lies at infinity.
    For the second generator, the sum of the points is taken.
    The imaginary part of each coordinate is omitted.
    """
    pone = list(plane[:,0])
    ptwo = list(plane[:,1])
    aone = [x.real for x in pone]
    atwo = [x.real + y.real for (x, y) in zip(pone, ptwo)]
    return (aone[1:], atwo[1:])
```

```
def boxrange(inlines, outlines):
    """
    Returns a list of three lists with the [min, max]
    values of each coordinate of each generator in the lists
    inlines and outlines.
    The ranges are adjusted for the particular real case.
    """
    fst = inlines[0][0]
```

(continues on next page)

(continued from previous page)

```
    result = {'xmin': fst[0], 'xmax': fst[0], \
              'ymin': fst[1], 'ymax': fst[1], \
              'zmin': fst[2], 'zmax': fst[2]}
    pts = [x for (x, y) in inlines] + [y for (x, y) in inlines] \
        + [x for (x, y) in outlines] + [y for (x, y) in outlines]
    print('the points :\n', pts)
    for point in pts:
        result['xmin'] = min(result['xmin'], point[0])
        result['ymin'] = min(result['ymin'], point[1])
        result['zmin'] = min(result['zmin'], point[2])
        result['xmax'] = max(result['xmax'], point[0])
        result['ymax'] = max(result['ymax'], point[1])
        result['zmax'] = max(result['zmax'], point[2])
    return ((result['xmin']+3, result['xmax']-3), \
            (result['ymin']+8, result['ymax']-11), \
            (result['zmin']+3, result['zmax']-5))
```

```
def inbox(point, lims):
    """
    Returns true if the coordinates of the point
    are in the box defined by the 3-tuple lims
    which contain the minima and maxima for the coordinates.
    """
    tol = 1.0e-8 # this is essential for roundoff
    (xlim, ylim, zlim) = lims
    if point[0] < xlim[0] - tol:
        return False
    elif point[0] > xlim[1] + tol:
        return False
    elif point[1] < ylim[0] - tol:
        return False
    elif point[1] > ylim[1] + tol:
        return False
    elif point[2] < zlim[0] - tol:
        return False
    elif point[2] > zlim[1] + tol:
        return False
    else:
        return True
```

```
def equal(pt1, pt2):
    """
    Returns true if the all coordinates of pt1 and pt2
    match up to a tolerance of 1.0e-10.
    """
    tol = 1.0e-8
    if abs(pt1[0] - pt2[0]) > tol:
        return False
    elif abs(pt1[1] - pt2[1]) > tol:
        return False
    elif abs(pt1[2] - pt2[2]) > tol:
```

(continues on next page)

(continued from previous page)

```
        return False
    return True
```

```
def isin(points, pnt):
    """
    Returns true if pnt belongs to the list points.
    """
    if len(points) == 0:
        return False
    else:
        for point in points:
            if equal(point, pnt):
                return True
        return False
```

```
def plot_line(axs, line, lims, color):
    """
    Plots the line defined as a tuple of two points,
    using the axis object in axs.
    The 3-tuple lims contains three lists with limits [min, max]
    for the x, y, and z coordinates.
    """
    (fst, snd) = line
    axs.set_xlabel('x')
    axs.set_ylabel('y')
    axs.set_zlabel('z')
    axs.set_xlim(lims[0])
    axs.set_ylim(lims[1])
    axs.set_zlim(lims[2])
    dir = (fst[0] - snd[0], fst[1] - snd[1], fst[2] - snd[2])
    result = []
    for k in range(3):
        fac = (lims[k][1]-fst[k])/dir[k]
        pnt = (fst[0] + fac*dir[0], fst[1] + fac*dir[1], fst[2] + fac*dir[2])
        if inbox(pnt, lims):
            if not isin(result, pnt): result.append(pnt)
    for k in range(3):
        fac = (lims[k][0]-fst[k])/dir[k]
        pnt = (fst[0] + fac*dir[0], fst[1] + fac*dir[1], fst[2] + fac*dir[2])
        if inbox(pnt, lims):
            if not isin(result, pnt): result.append(pnt)
    (one, two) = (result[0], result[1])
    # axs.plot([fst[0], snd[0]], [fst[1], snd[1]], [fst[2], snd[2]], 'bo')
    # axs.plot([one[0], two[0]], [one[1], two[1]], [one[2], two[2]], 'ro')
    axs.plot([one[0], two[0]], [one[1], two[1]], [one[2], two[2]], color)
    plt.savefig('fourlinesfig1')
```

```
def plot_lines(inlines, outlines, points, lims):
    """
    Generates coordinates of the points in a random line
    and then plots this line.  The intersection points are
```

(continues on next page)

(continued from previous page)

```
    in the list points and limits for the bounding box in lims
    """
    fig = plt.figure()
    axs = fig.add_subplot(111, projection='3d')
    for line in inlines:
        plot_line(axs, line, lims, 'b')
    for line in outlines:
        plot_line(axs, line, lims, 'r')
    for point in points:
        axs.plot([point[0]], [point[1]], [point[2]], 'ro')
    axs.view_init(azim=5, elev=20)
    plt.show()
    plt.savefig('fourlinesfig2')
```

```
def intersection_point(apl, bpl, check=True):
    """
    Given in apl the two points that define a line
    and in bpl the two points that define another line,
    returns the intersection point.
    If check, then additional tests are done
    and the outcome of the tests is written to screen.
    """
    (apt, bpt) = apl
    (cpt, dpt) = bpl
    mat = array([[apt[0], bpt[0], -cpt[0]], \
                 [apt[1], bpt[1], -cpt[1]], \
                 [apt[2], bpt[2], -cpt[2]]])
    rhs = array([[dpt[0]], [dpt[1]], [dpt[2]]])
    sol = solve(mat, rhs)
    cff = list(sol[:,0])
    csm = cff[0] + cff[1]
    result = ((cff[0]*apt[0] + cff[1]*bpt[0])/csm, \
              (cff[0]*apt[1] + cff[1]*bpt[1])/csm, \
              (cff[0]*apt[2] + cff[1]*bpt[2])/csm)
    if check:
        csm = cff[2] + 1.0
        verify = ((cff[2]*cpt[0] + dpt[0])/csm, \
                  (cff[2]*cpt[1] + dpt[1])/csm, \
                  (cff[2]*cpt[2] + dpt[2])/csm)
        print('the solution :\\n', result)
        print('the solution verified :\\n', verify)
        res = matrix(rhs) - matrix(mat)*matrix(sol)
        print('the residual :\n', res)
    return result
```

```
def intersection_points(ipl, opl):
    """
    Returns the list of intersection points between
    the input planes in ipl and the output planes in opl.
    """
    result = []
```

(continues on next page)

(continued from previous page)

```
    for inplane in ipl:
        for outplane in opl:
            result.append(intersection_point(inplane, outplane))
    return result
```

```
def show_planes(ipl, opl):
    """
    Shows the input and the output planes.
    """
    (inlines, outlines) = ([], [])
    for plane in ipl:
        inlines.append(input_generators(plane))
    for plane in opl:
        outlines.append(output_generators(plane))
    print('The generators of the input lines :')
    for line in inlines:
        print(line)
    print('The generators of the output lines :')
    for line in outlines:
        print(line)
    brg = boxrange(inlines, outlines)
    print('the range:', brg)
    intpts = intersection_points(inlines, outlines)
    print('the intersection points :')
    for point in intpts:
        print(point)
    plot_lines(inlines, outlines, intpts, brg)
    plt.savefig('fourlinesfig3')
```

We end up with an interactive backend for the 3d plot.

```
%matplotlib widget
show_planes(oscp, otp2)
```

produces the following output:

```
The generators of the input lines :
([-0.3846269613221122, -0.7041242012482366, 0.9262772651610497], [5.356155982058531, -9.
↪53636379773747, -6.104719131981401])
([-0.4924082638753003, -0.5150682033346254, 0.9966559434380463], [-6.2997304815421655,␣
↪10.923225600528575, 1.5246914154709972])
([0.6258059455871108, -0.2167338369356443, -0.8970725931155779], [2.1821835337633937, 3.
↪6792275561013374, 1.7481420529767318])
([0.13188052906200864, -0.9652150521086493, -0.3864666725234145], [-2.8596260453517486, -
↪2.5433009310133032, 7.963696648872165])
The generators of the output lines :
([-0.0284638025557899, -1.06771882518925, 0.0], [0.97153619744421, -1.187067575737539, -
↪4.99706612461873])
([-0.055273486968536, -0.660558824288729, 0.0], [0.944726513031464, -0.779907574837019, -
↪2.57330433323918])
the points :
[[-0.3846269613221122, -0.7041242012482366, 0.9262772651610497], [-0.4924082638753003, -
↪0.5150682033346254, 0.9966559434380463], [0.6258059455871108, -0.2167338369356443, -0.
↪8970725931155779], [0.13188052906200864, -0.9652150521086493, -0.3864666725234145], [5.
↪356155982058531, -9.53636379773747, -6.104719131981401], [-6.2997304815421655, 10.
↪923225600528575, 1.5246914154709972], [2.1821835337633937, 3.6792275561013374, 1.
↪7481420529767318], [-2.8596260453517486, -2.5433009310133032, 7.963696648872165], [-0.
↪0284638025557899, -1.06771882518925, 0.0], [-0.055273486968536, -0.660558824288729, 0.
↪0], [0.97153619744421, -1.187067575737539, -4.99706612461873], [0.944726513031464, -0.
```

(continues on next page)

(continued from previous page)

```
the range: ((-3.2997304815421655, 2.3561559820585307), (-1.5363637977374704, -0.
↪07677439947142517), (-3.104719131981401, 2.9636966488721654))
the solution :
 (-0.15837537533646365, -1.052214041296111, 0.6491767195382462)
the solution verified :
 (-0.15837537533646406, -1.0522140412961136, 0.6491767195382475)
the residual :
 [[4.44089210e-16]
 [1.11022302e-15]
 [0.00000000e+00]]
the solution :
 (-0.4430230234123302, -0.6142814015884848, 0.9977975623422988)
the solution verified :
 (-0.44302302341232946, -0.6142814015884835, 0.997797562342297)
the residual :
 [[ 0.00000000e+00]
 [-2.22044605e-16]
 [ 0.00000000e+00]]
the solution :
 (-0.2236498742909531, -1.0444236114032255, 0.9753577070651858)
the solution verified :
 (-0.22364987429095395, -1.0444236114032293, 0.9753577070651895)
the residual :
 [[-1.11022302e-16]
 [-6.66133815e-16]
 [ 0.00000000e+00]]
the solution :
 (-0.441973240857878, -0.6144066918247044, 0.9950961523459683)
the solution verified :
 (-0.44197324085787826, -0.6144066918247048, 0.9950961523459688)
the residual :
 [[1.11022302e-16]
 [2.22044605e-16]
 [0.00000000e+00]]
the solution :
 (0.2715464337673154, -1.1035246720461052, -1.4991709889690488)
the solution verified :
 (0.27154643376731663, -1.1035246720461096, -1.4991709889690552)
the residual :
 [[1.11022302e-16]
 [2.22044605e-16]
 [0.00000000e+00]]
the solution :
 (0.42557851238329614, -0.7179479096100174, -1.2373785335787928)
the solution verified :
 (0.42557851238329597, -0.7179479096100173, -1.2373785335787926)
the residual :
 [[-1.11022302e-16]
 [ 0.00000000e+00]
 [ 0.00000000e+00]]
the solution :
 (-0.056164218290926694, -1.0644128151815966, 0.1384208091079073)
```

(continues on next page)

(continued from previous page)

```
the solution verified :
 (-0.05616421829092654, -1.0644128151815933, 0.1384208091079069)
the residual :
 [[ 6.66133815e-16]
 [-2.44249065e-15]
 [-1.77635684e-15]]
the solution :
 (0.5683194604437922, -0.7349838634131002, -1.6046944337535327)
the solution verified :
 (0.5683194604438059, -0.7349838634131174, -1.6046944337535711)
the residual :
 [[ 1.11022302e-16]
 [ 3.33066907e-16]
 [-4.44089210e-16]]
the intersection points :
(-0.15837537533646365, -1.052214041296111, 0.6491767195382462)
(-0.4430230234123302, -0.6142814015884848, 0.9977975623422988)
(-0.2236498742909531, -1.0444236114032255, 0.9753577070651858)
(-0.441973240857878, -0.6144066918247044, 0.9950961523459683)
(0.2715464337673154, -1.1035246720461052, -1.4991709889690488)
(0.42557851238329614, -0.7179479096100174, -1.2373785335787928)
(-0.05616421829092694, -1.0644128151815966, 0.1384208091079073)
(0.5683194604437922, -0.7349838634131002, -1.6046944337535327)
```

The code produces Fig. 2.6.

2.4 The Circle Problem of Apollonius

The circle problem of Apollonius has the following input/output specification:

Given three circles, find all circles that are tangent to the given circles.

2.4.1 the polynomial systems

Without loss of generality, we take the first circle to be the unit circle, centered at (0, 0) and with radius 1. The origin of the second circle lies on the first coordinate axis, so its center has coordinates (`c2x`, 0) and radius `r2`. The third circle has center (`c3x`, `c3y`) and radius `r3`. So there are five parameters in this problem: `c2x`, `r2`, `c3x`, `c3y`, and `r3`. Values for the five parameters are defined by the first five equations. The next three equations determine the center (`x`, `y`) and the radius `r` of the circle which touches the three given circles. The condition on the center of the touching circle is that its distance to the center of the given circle is either the difference or the sum of the radii of both circles. So we arrive at eight polynomial systems.

The problem formulation is coded in the function `polynomials`.

```
def polynomials(c2x, r2, c3x, c3y, r3):
    """
    On input are the five parameters of the circle problem of Apollonius:
    c2x : the x-coordinate of the center of the second circle,
    r2 : the radius of the second circle,
    c3x : the x-coordinate of the center of the third circle,
    c3y : the y-coordinate of the center of the third circle,
```

(continues on next page)

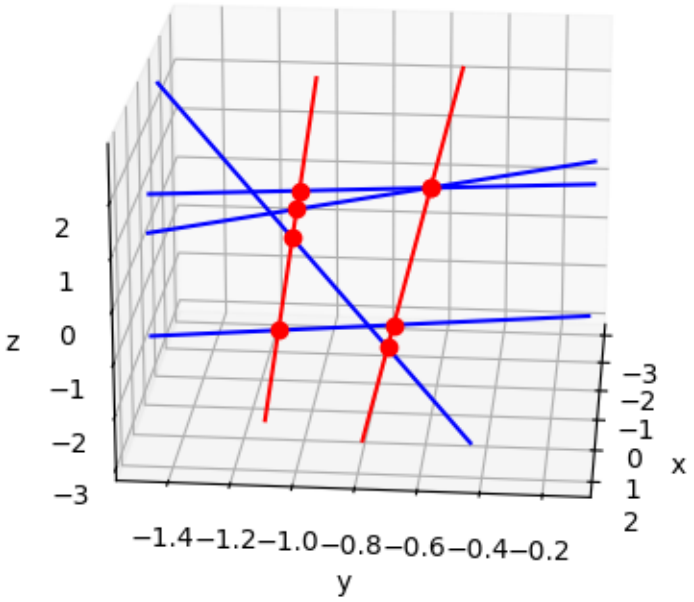


Fig. 2.6: Two lines meeting four given lines.

(continued from previous page)

```
    r3 : the radius of the third circle.
    Returns a list of lists.  Each list contains a polynomial system.
    Solutions to each polynomial system define center (x, y) and radius r
    of a circle touching three given circles.
    """
    e1m = 'x^2 + y^2 - (r-1)^2;'
    e1p = 'x^2 + y^2 - (r+1)^2;'
    e2m = '(x-%.15f)^2 + y^2 - (r-%.15f)^2;' % (c2x, r2)
    e2p = '(x-%.15f)^2 + y^2 - (r+%.15f)^2;' % (c2x, r2)
    e3m = '(x-%.15f)^2 + (y-%.15f)^2 - (r-%.15f)^2;' % (c3x, c3y, r3)
    e3p = '(x-%.15f)^2 + (y-%.15f)^2 - (r+%.15f)^2;' % (c3x, c3y, r3)
    eqs0 = [e1m,e2m,e3m]
    eqs1 = [e1m,e2m,e3p]
    eqs2 = [e1m,e2p,e3m]
    eqs3 = [e1m,e2p,e3p]
    eqs4 = [e1p,e2m,e3m]
    eqs5 = [e1p,e2m,e3p]
    eqs6 = [e1p,e2p,e3m]
    eqs7 = [e1p,e2p,e3p]
    return [eqs0,eqs1,eqs2,eqs3,eqs4,eqs5,eqs6,eqs7]
```

As an example of a general problem, the center of the second circle is at `(2, 0)`, with radius `2/3`, and the third circle is centered at `(1, 1)`, with a radius of `1/3`.

Let us look at the eight polynomial systems, computed as the output of the function `polynomials`.

```
general_problem = polynomials(2, 2.0/3, 1, 1, 1.0/3)
for pols in general_problem:
    print(pols)
```

The eight polynomial systems are shown below:

```
['x^2 + y^2 - (r-1)^2;', '(x-2.000000000000000)^2 + y^2 - (r-0.666666666666667)^2;', '(x-
↪1.000000000000000)^2 + (y-1.000000000000000)^2 - (r-0.333333333333333)^2;']
['x^2 + y^2 - (r-1)^2;', '(x-2.000000000000000)^2 + y^2 - (r-0.666666666666667)^2;', '(x-
↪1.000000000000000)^2 + (y-1.000000000000000)^2 - (r+0.333333333333333)^2;']
['x^2 + y^2 - (r-1)^2;', '(x-2.000000000000000)^2 + y^2 - (r+0.666666666666667)^2;', '(x-
↪1.000000000000000)^2 + (y-1.000000000000000)^2 - (r-0.333333333333333)^2;']
['x^2 + y^2 - (r-1)^2;', '(x-2.000000000000000)^2 + y^2 - (r+0.666666666666667)^2;', '(x-
↪1.000000000000000)^2 + (y-1.000000000000000)^2 - (r+0.333333333333333)^2;']
['x^2 + y^2 - (r+1)^2;', '(x-2.000000000000000)^2 + y^2 - (r-0.666666666666667)^2;', '(x-
↪1.000000000000000)^2 + (y-1.000000000000000)^2 - (r-0.333333333333333)^2;']
['x^2 + y^2 - (r+1)^2;', '(x-2.000000000000000)^2 + y^2 - (r-0.666666666666667)^2;', '(x-
↪1.000000000000000)^2 + (y-1.000000000000000)^2 - (r+0.333333333333333)^2;']
['x^2 + y^2 - (r+1)^2;', '(x-2.000000000000000)^2 + y^2 - (r+0.666666666666667)^2;', '(x-
↪1.000000000000000)^2 + (y-1.000000000000000)^2 - (r-0.333333333333333)^2;']
['x^2 + y^2 - (r+1)^2;', '(x-2.000000000000000)^2 + y^2 - (r+0.666666666666667)^2;', '(x-
↪1.000000000000000)^2 + (y-1.000000000000000)^2 - (r+0.333333333333333)^2;']
```

2.4.2 plotting circles

The package `matplotlib` has primitives to define circles.

```
import matplotlib.pyplot as plt
```

The input to the three given circles of the general problem is codified in the list of tuples set below.

```
crcdata = [((0, 0), 1), ((2, 0), 2.0/3), ((1, 1), 1.0/3)]
```

The input circles will be shown as blue disks. Let us then render our general configuration.

```
(xa, xb, ya, yb) = (-2, 4, -2, 3)
```

The code to make Fig. 2.7 is below:

```
fig = plt.figure()
axs = fig.add_subplot(111, aspect='equal')
for (center, radius) in crcdata:
    crc = plt.Circle(center, radius, edgecolor='blue', facecolor='blue')
    axs.add_patch(crc)
plt.axis([xa, xb, ya, yb])
fig.canvas.draw()
```

2.4.3 solving polynomial systems

To solve the polynomial systems, we apply the blackbox solver.

```
from phcpy.solver import solve
```

and we need some functions to extract the real solutions.

```
from phcpy.solutions import strsol2dict, is_real
```

The `solve4circles` calls the solver on the polynomial systems of the problem.

```
def solve4circles(syst, verbose=True):
    """
    Given in syst is a list of polynomial systems.
    Returns a list of tuples.  Each tuple in the list of return
    consists of the coordinates of the center and the radius of
    a circle touching the three given circles.
    """
    (circle, eqscnt) = (0, 0)
    result = []
    for eqs in syst:
        eqscnt = eqscnt + 1
        if verbose:
            print('solving system', eqscnt, ':')
            for pol in eqs:
                print(pol)
        sols = solve(eqs)
        if verbose:
```

(continues on next page)

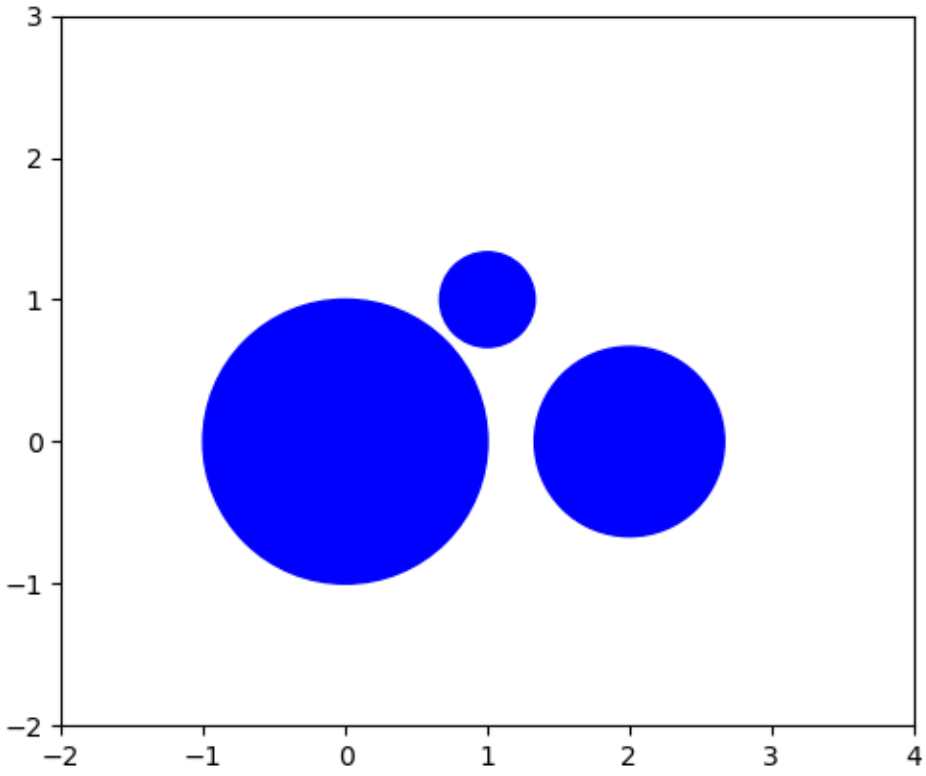


Fig. 2.7: Three input circles.

(continued from previous page)

```
            print('system', eqscnt, 'has', len(sols), 'solutions')
        for sol in sols:
            if is_real(sol, 1.0e-8):
                soldic = strsol2dict(sol)
                if soldic['r'].real > 0:
                    circle = circle + 1
                    ctr = (soldic['x'].real, soldic['y'].real)
                    rad = soldic['r'].real
                    result.append((ctr, rad))
                    if verbose:
                        print('solution circle', circle)
                        print('center =', ctr)
                        print('radius =', rad)
    return result
```

The function `solve4circles` puts the solutions of the polynomial system in the format of our problem. Each solution is a circle, represented by a tuple of the coordinates of the center and the radius of the circle.

```
sols = solve4circles(general_problem)
```

has as output

```
solving system 1 :
x^2 + y^2 - (r-1)^2;
(x-2.000000000000000)^2 + y^2 - (r-0.666666666666667)^2;
(x-1.000000000000000)^2 + (y-1.000000000000000)^2 - (r-0.333333333333333)^2;
system 1 has 2 solutions
solution circle 1
center = (0.792160611810177, -0.734629275680581)
radius = 2.08036966247227
solving system 2 :
x^2 + y^2 - (r-1)^2;
(x-2.000000000000000)^2 + y^2 - (r-0.666666666666667)^2;
(x-1.000000000000000)^2 + (y-1.000000000000000)^2 - (r+0.333333333333333)^2;
system 2 has 2 solutions
solving system 3 :
x^2 + y^2 - (r-1)^2;
(x-2.000000000000000)^2 + y^2 - (r+0.666666666666667)^2;
(x-1.000000000000000)^2 + (y-1.000000000000000)^2 - (r-0.333333333333333)^2;
system 3 has 2 solutions
solution circle 2
center = (-0.200806137165905, 0.573494560766514)
radius = 1.60763403126575
solving system 4 :
x^2 + y^2 - (r-1)^2;
(x-2.000000000000000)^2 + y^2 - (r+0.666666666666667)^2;
(x-1.000000000000000)^2 + (y-1.000000000000000)^2 - (r+0.333333333333333)^2;
system 4 has 2 solutions
solution circle 3
center = (-0.0193166119185703, -0.389367744928919)
radius = 1.38984660096895
solving system 5 :
```

(continues on next page)

(continued from previous page)

```
x^2 + y^2 - (r+1)^2;
(x-2.000000000000000)^2 + y^2 - (r-0.666666666666667)^2;
(x-1.000000000000000)^2 + (y-1.000000000000000)^2 - (r-0.333333333333333)^2;
system 5 has 2 solutions
solution circle 4
center = (5.35264994525194, 2.83381218937338)
radius = 5.05651326763565
solving system 6 :
x^2 + y^2 - (r+1)^2;
(x-2.000000000000000)^2 + y^2 - (r-0.666666666666667)^2;
(x-1.000000000000000)^2 + (y-1.000000000000000)^2 - (r+0.333333333333333)^2;
system 6 has 2 solutions
solution circle 5
center = (1.86747280383257, 0.159838772566819)
radius = 0.874300697932419
solving system 7 :
x^2 + y^2 - (r+1)^2;
(x-2.000000000000000)^2 + y^2 - (r+0.666666666666667)^2;
(x-1.000000000000000)^2 + (y-1.000000000000000)^2 - (r-0.333333333333333)^2;
system 7 has 2 solutions
solution circle 6
center = (1.43293571744453, 2.36388335544507)
radius = 1.76428097133387
solution circle 7
center = (1.23373094922213, 0.96944997788827)
radius = 0.56905236199947
solving system 8 :
x^2 + y^2 - (r+1)^2;
(x-2.000000000000000)^2 + y^2 - (r+0.666666666666667)^2;
(x-1.000000000000000)^2 + (y-1.000000000000000)^2 - (r+0.333333333333333)^2;
system 8 has 2 solutions
solution circle 8
center = (1.1821983625488, 0.435483976535281)
radius = 0.25985684195945
```

As a summary, let us print the solution circles.

```
for (idx, circle) in enumerate(sols):
    print('Circle', idx+1, ':', circle)
```

```
Circle 1 : ((0.792160611810177, -0.734629275680581), 2.08036966247227)
Circle 2 : ((-0.200806137165905, 0.573494560766514), 1.60763403126575)
Circle 3 : ((-0.0193166119185703, -0.389367744928919), 1.38984660096895)
Circle 4 : ((5.35264994525194, 2.83381218937338), 5.05651326763565)
Circle 5 : ((1.86747280383257, 0.159838772566819), 0.874300697932419)
Circle 6 : ((1.43293571744453, 2.36388335544507), 1.76428097133387)
Circle 7 : ((1.23373094922213, 0.96944997788827), 0.56905236199947)
Circle 8 : ((1.1821983625488, 0.435483976535281), 0.25985684195945)
```

Observe that we have a constellation where all eight touching circles have real coordinates as centers and a positive radius.

In Fig. 2.8 the given circles are plotted as blue disks, while the eight solution circles are plotted in red, done by the

code below.

```
fig = plt.figure()
axs = fig.add_subplot(111, aspect='equal')
for (center, radius) in crcdata:
    crc = plt.Circle(center, radius, edgecolor='blue', facecolor='blue')
    axs.add_patch(crc)
for (center, radius) in sols:
    crc = plt.Circle(center, radius, edgecolor='red', facecolor='none')
    axs.add_patch(crc)
plt.axis([xa, xb, ya, yb])
fig.canvas.draw()
```

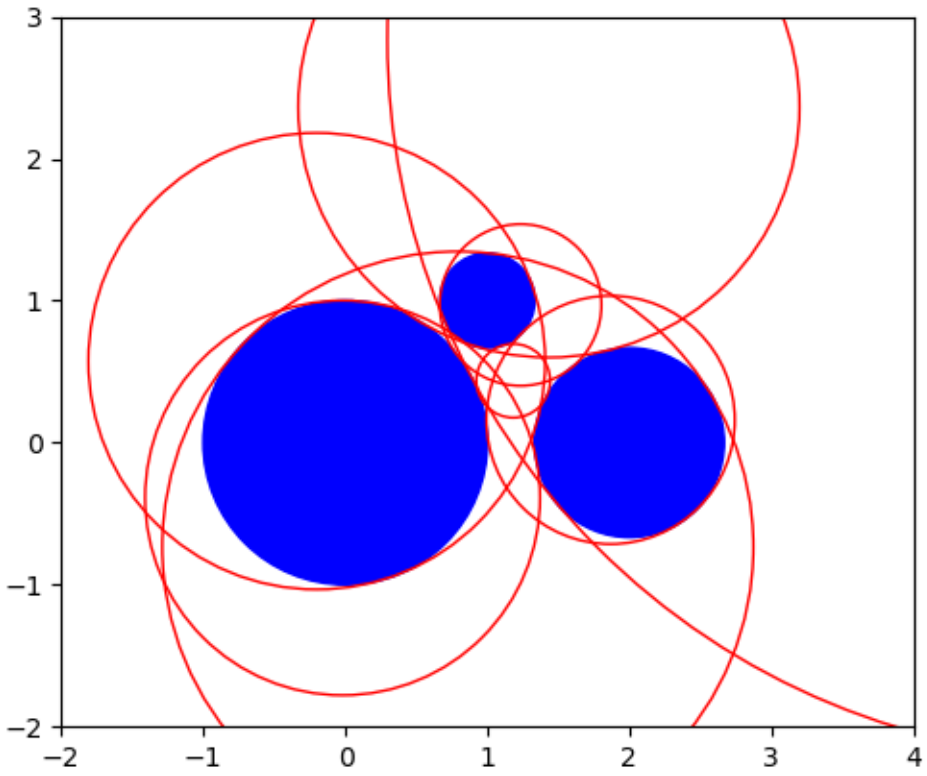


Fig. 2.8: Eight circles touching three given circles.

2.4.4 a special problem

In a special configuration of three circles, the three circles are mutually touching each other.

```
from math import sqrt
height = sqrt(3)
```

The output of

```
special_problem = polynomials(2, 1, 1, height, 1)
for pols in special_problem:
    print(pols)
```

is the following list of eight polynomial systems:

```
['x^2 + y^2 - (r-1)^2;', '(x-2.000000000000000)^2 + y^2 - (r-1.000000000000000)^2;', '(x-
↪1.000000000000000)^2 + (y-1.732050807568877)^2 - (r-1.000000000000000)^2;']
['x^2 + y^2 - (r-1)^2;', '(x-2.000000000000000)^2 + y^2 - (r-1.000000000000000)^2;', '(x-
↪1.000000000000000)^2 + (y-1.732050807568877)^2 - (r+1.000000000000000)^2;']
['x^2 + y^2 - (r-1)^2;', '(x-2.000000000000000)^2 + y^2 - (r+1.000000000000000)^2;', '(x-
↪1.000000000000000)^2 + (y-1.732050807568877)^2 - (r-1.000000000000000)^2;']
['x^2 + y^2 - (r-1)^2;', '(x-2.000000000000000)^2 + y^2 - (r+1.000000000000000)^2;', '(x-
↪1.000000000000000)^2 + (y-1.732050807568877)^2 - (r+1.000000000000000)^2;']
['x^2 + y^2 - (r+1)^2;', '(x-2.000000000000000)^2 + y^2 - (r-1.000000000000000)^2;', '(x-
↪1.000000000000000)^2 + (y-1.732050807568877)^2 - (r-1.000000000000000)^2;']
['x^2 + y^2 - (r+1)^2;', '(x-2.000000000000000)^2 + y^2 - (r-1.000000000000000)^2;', '(x-
↪1.000000000000000)^2 + (y-1.732050807568877)^2 - (r+1.000000000000000)^2;']
['x^2 + y^2 - (r+1)^2;', '(x-2.000000000000000)^2 + y^2 - (r+1.000000000000000)^2;', '(x-
↪1.000000000000000)^2 + (y-1.732050807568877)^2 - (r-1.000000000000000)^2;']
['x^2 + y^2 - (r+1)^2;', '(x-2.000000000000000)^2 + y^2 - (r+1.000000000000000)^2;', '(x-
↪1.000000000000000)^2 + (y-1.732050807568877)^2 - (r+1.000000000000000)^2;']
```

```
specialinput = [((0, 0), 1), ((2, 0), 1), ((1, height), 1)]
(xa, xb, ya, yb) = (-2, 4, -2, 4)
```

The code to show the special input is

```
fig = plt.figure()
axs = fig.add_subplot(111, aspect='equal')
for (center, radius) in specialinput:
    crc = plt.Circle(center, radius, edgecolor='blue', facecolor='blue')
    axs.add_patch(crc)
plt.axis([xa, xb, ya, yb])
fig.canvas.draw()
```

which produces Fig. 2.9.

The output of

```
specialsols = solve4circles(special_problem)
```

is

```
solving system 1 :
x^2 + y^2 - (r-1)^2;
```

(continues on next page)

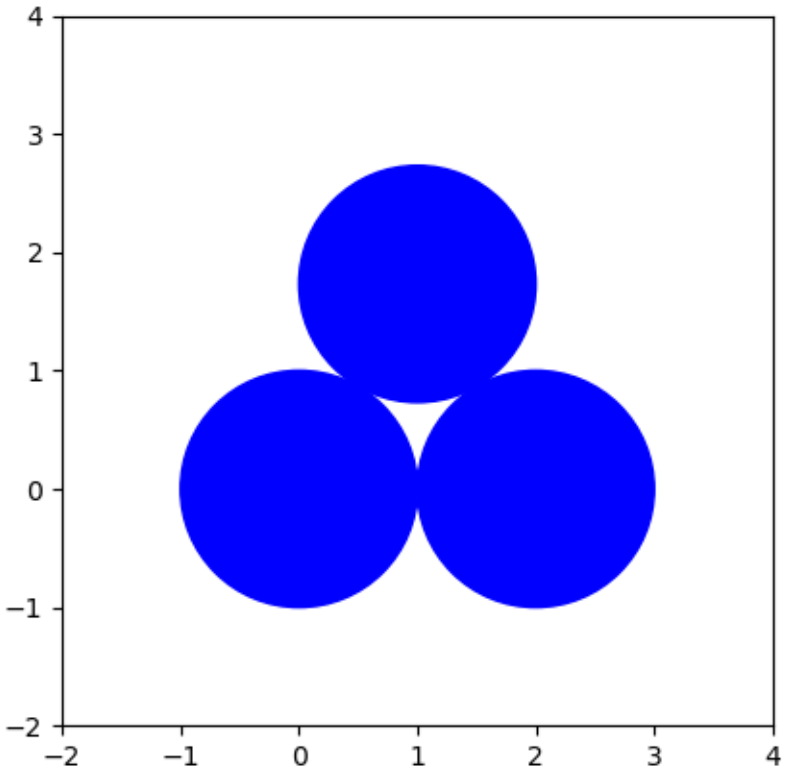


Fig. 2.9: Three touching input circles.

(continued from previous page)

```
(x-2.000000000000000)^2 + y^2 - (r-1.000000000000000)^2;
(x-1.000000000000000)^2 + (y-1.732050807568877)^2 - (r-1.000000000000000)^2;
system 1 has 2 solutions
solution circle 1
center = (1.0, 0.577350269189626)
radius = 2.15470053837925
solving system 2 :
x^2 + y^2 - (r-1)^2;
(x-2.000000000000000)^2 + y^2 - (r-1.000000000000000)^2;
(x-1.000000000000000)^2 + (y-1.732050807568877)^2 - (r+1.000000000000000)^2;
system 2 has 1 solutions
solving system 3 :
x^2 + y^2 - (r-1)^2;
(x-2.000000000000000)^2 + y^2 - (r+1.000000000000000)^2;
(x-1.000000000000000)^2 + (y-1.732050807568877)^2 - (r-1.000000000000000)^2;
system 3 has 1 solutions
solving system 4 :
x^2 + y^2 - (r-1)^2;
(x-2.000000000000000)^2 + y^2 - (r+1.000000000000000)^2;
(x-1.000000000000000)^2 + (y-1.732050807568877)^2 - (r+1.000000000000000)^2;
system 4 has 1 solutions
solution circle 2
center = (2.89107059865923e-16, 1.15377761182971e-16)
radius = 1.0
solving system 5 :
x^2 + y^2 - (r+1)^2;
(x-2.000000000000000)^2 + y^2 - (r-1.000000000000000)^2;
(x-1.000000000000000)^2 + (y-1.732050807568877)^2 - (r-1.000000000000000)^2;
system 5 has 1 solutions
solving system 6 :
x^2 + y^2 - (r+1)^2;
(x-2.000000000000000)^2 + y^2 - (r-1.000000000000000)^2;
(x-1.000000000000000)^2 + (y-1.732050807568877)^2 - (r+1.000000000000000)^2;
system 6 has 1 solutions
solution circle 3
center = (2.0, 7.69185074553436e-17)
radius = 1.0
solving system 7 :
x^2 + y^2 - (r+1)^2;
(x-2.000000000000000)^2 + y^2 - (r+1.000000000000000)^2;
(x-1.000000000000000)^2 + (y-1.732050807568877)^2 - (r-1.000000000000000)^2;
system 7 has 1 solutions
solution circle 4
center = (1.0, 1.73205080756888)
radius = 0.999999999999999
solving system 8 :
x^2 + y^2 - (r+1)^2;
(x-2.000000000000000)^2 + y^2 - (r+1.000000000000000)^2;
(x-1.000000000000000)^2 + (y-1.732050807568877)^2 - (r+1.000000000000000)^2;
system 8 has 2 solutions
solution circle 5
center = (1.0, 0.577350269189626)
```

(continues on next page)

(continued from previous page)

```
radius = 0.154700538379251
```

Let us look closer at the solutions :

```
for (idx, circle) in enumerate(specialsols):
    print('Circle', idx+1, ':', circle)
```

```
Circle 1 : ((1.0, 0.577350269189626), 2.15470053837925)
Circle 2 : ((2.89107059865923e-16, 1.15377761182971e-16), 1.0)
Circle 3 : ((2.0, 7.69185074553436e-17), 1.0)
Circle 4 : ((1.0, 1.73205080756888), 0.999999999999999)
Circle 5 : ((1.0, 0.577350269189626), 0.154700538379251)
```

We have five solutions? Not eight?

The code for the next plot is in

```
fig = plt.figure()
axs = fig.add_subplot(111, aspect='equal')
for (center, radius) in specialinput:
    crc = plt.Circle(center, radius, edgecolor='blue', facecolor='blue')
    axs.add_patch(crc)
for (center, radius) in specialsols:
    crc = plt.Circle(center, radius, edgecolor='red', facecolor='none')
    axs.add_patch(crc)
plt.axis([xa, xb, ya, yb])
fig.canvas.draw()
```

The plot in Fig. 2.10 shows that the input circles are solutions as well.

2.4.5 a perturbed problem

Consider a small perturbation of a special configuration of three circles, where the three circles are mutually touching each other.

```
perturbedinput = [((0, 0), 1), ((2.05, 0), 1), ((1.025, height+0.025), 1)]
perturbed_problem = polynomials(2.05, 1, 1.025, height+0.025, 1)\n",
perturbedsols = solve4circles(perturbed_problem)"
```

produces the following output :

```
solving system 1 :
x^2 + y^2 - (r-1)^2;
(x-2.050000000000000)^2 + y^2 - (r-1.000000000000000)^2;
(x-1.025000000000000)^2 + (y-1.757050807568877)^2 - (r-1.000000000000000)^2;
system 1 has 2 solutions
solution circle 1
center = (1.025, 0.579551408418395)
radius = 2.17749939915048
solving system 2 :
x^2 + y^2 - (r-1)^2;
(x-2.050000000000000)^2 + y^2 - (r-1.000000000000000)^2;
```

(continues on next page)

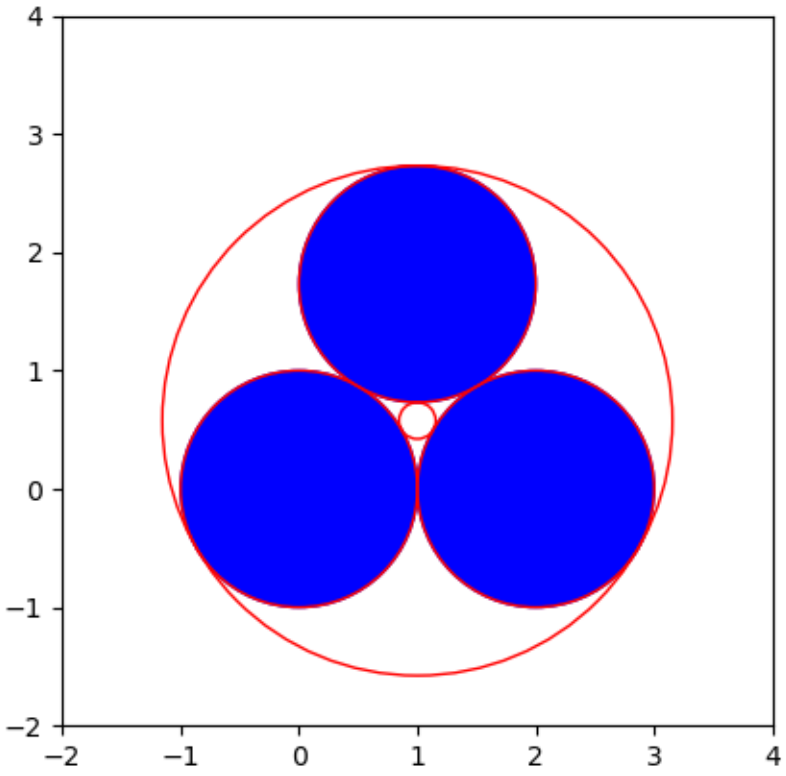


Fig. 2.10: All circles touching three given touching circles.

(continued from previous page)

```
(x-1.025000000000000)^2 + (y-1.757050807568877)^2 - (r+1.000000000000000)^2;
system 2 has 2 solutions
solving system 3 :
x^2 + y^2 - (r-1)^2;
(x-2.050000000000000)^2 + y^2 - (r+1.000000000000000)^2;
(x-1.025000000000000)^2 + (y-1.757050807568877)^2 - (r-1.000000000000000)^2;
system 3 has 2 solutions
solving system 4 :
x^2 + y^2 - (r-1)^2;
(x-2.050000000000000)^2 + y^2 - (r+1.000000000000000)^2;
(x-1.025000000000000)^2 + (y-1.757050807568877)^2 - (r+1.000000000000000)^2;
system 4 has 2 solutions
solution circle 2
center = (0.0248497799767383, -0.00390011791639834)
radius = 1.02515397552384
solution circle 3
center = (-0.309008334843067, -0.198660887619915)
radius = 1.36735854321414
solving system 5 :
x^2 + y^2 - (r+1)^2;
(x-2.050000000000000)^2 + y^2 - (r-1.000000000000000)^2;
(x-1.025000000000000)^2 + (y-1.757050807568877)^2 - (r-1.000000000000000)^2;
system 5 has 2 solutions
solving system 6 :
x^2 + y^2 - (r+1)^2;
(x-2.050000000000000)^2 + y^2 - (r-1.000000000000000)^2;
(x-1.025000000000000)^2 + (y-1.757050807568877)^2 - (r+1.000000000000000)^2;
system 6 has 2 solutions
solution circle 4
center = (2.35900833484306, -0.19866088761991)
radius = 1.36735854321413
solution circle 5
center = (2.02515022002328, -0.00390011791640729)
radius = 1.02515397552386
solving system 7 :
x^2 + y^2 - (r+1)^2;
(x-2.050000000000000)^2 + y^2 - (r+1.000000000000000)^2;
(x-1.025000000000000)^2 + (y-1.757050807568877)^2 - (r-1.000000000000000)^2;
system 7 has 2 solutions
solution circle 6
center = (1.025, 1.73870496299037)
radius = 1.01834584457851
solution circle 7
center = (1.025, 2.04075732867107)
radius = 1.28370652110219
solving system 8 :
x^2 + y^2 - (r+1)^2;
(x-2.050000000000000)^2 + y^2 - (r+1.000000000000000)^2;
(x-1.025000000000000)^2 + (y-1.757050807568877)^2 - (r+1.000000000000000)^2;
system 8 has 2 solutions
solution circle 8
center = (1.025, 0.579551408418395)
```

(continues on next page)

(continued from previous page)

```
radius = 0.177499399150482
```

Let us look at the solution circles :

```
for (idx, circle) in enumerate(perturbedsols):
    print('circle', idx+1, ':', circle)
```

```
circle 1 : ((1.025, 0.579551408418395), 2.17749939915048)
circle 2 : ((0.0248497799767383, -0.00390011791639834), 1.02515397552384)
circle 3 : ((-0.309008334843067, -0.198660887619915), 1.36735854321414)
circle 4 : ((2.35900833484306, -0.19866088761991), 1.36735854321413)
circle 5 : ((2.02515022002328, -0.00390011791640729), 1.02515397552386)
circle 6 : ((1.025, 1.73870496299037), 1.01834584457851)
circle 7 : ((1.025, 2.04075732867107), 1.28370652110219)
circle 8 : ((1.025, 0.579551408418395), 0.177499399150482)
```

Fig. 2.11 is made by the code below:

```
fig = plt.figure()
axs = fig.add_subplot(111, aspect='equal')
for (center, radius) in perturbedinput:
    crc = plt.Circle(center, radius, edgecolor='blue', facecolor='blue')
    axs.add_patch(crc)
for (center, radius) in perturbedsols:
    crc = plt.Circle(center, radius, edgecolor='red', facecolor='none')
    axs.add_patch(crc)
plt.axis([xa, xb, ya, yb])
fig.canvas.draw()
```

The solution to the perturbed problem allows to account for the number five as the number of touching circles of the special problem: the original circles had to be counted twice, as their multiplicity equals two. And so we thus have $3 \times 2 + 2 = 8$.

2.5 All Lines Tangent to Four Spheres

Consider all tangent lines to four mutually touching spheres.

The original formulation as polynomial system came from Cassiano Durand, then at the CS department in Purdue. The positioning of the centers of the spheres, each with radius 0.5 at the vertices of a tetrahedron came from Thorsten Theobald, then at TU Muenich. The centers of the four spheres are

$$c_1 = (0,0,0), \quad c_2 = (1,0,0), \quad c_3 = (1/2, \sqrt{3}/2, 0), \quad c_4 = (1/2, \sqrt{3}/6, \sqrt{6}/3).$$

Let $t = (x_0, x_1, x_2)$ be the tangent vector and $m = (x_3, x_4, x_5)$ the moment vector.

The first equation is $\|t\| = 1$, the second $m \cdot t = 0$, the other equations are $\|m - c_i \times t\|^2 - r^2 = 0$ where the radius $r = 1/2$.

```
from sympy import var, sqrt
from sympy.vector import CoordSys3D, Vector
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

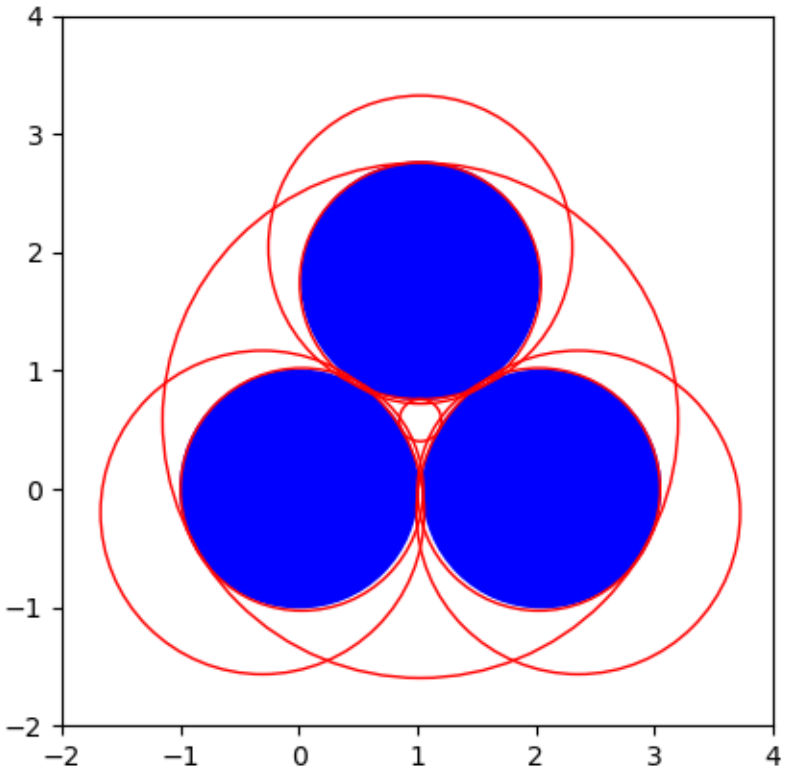


Fig. 2.11: The solution to the perturbed problem

(continued from previous page)

```
from phcpy.solver import solve
from phcpy.solutions import coordinates
```

2.5.1 centers and radii

Choices of the centers and radii of four mutually tangent spheres are defined here.

```
ctr1 = (0, 0, 0)
ctr2 = (1, 0, 0)
ctr3 = (0.5, sqrt(3.0)/2, 0)
ctr4 = (0.5, sqrt(3.0)/6, sqrt(6.0)/3)
radius = 0.5
centers = [ctr1, ctr2, ctr3, ctr4]
```

The choices were made for the suitability of the plot. Other choices can be found in the paper by Frank Sottile and Thorsten Theobald: **Line problems in nonlinear computational geometry**. In *Computational Geometry - Twenty Years Later*, pages 411-432, edited by J.E. Goodman, J. Pach, and R. Pollack, AMS, 2008.

2.5.2 formulating the equations

We need some vector calculus, done with `sympy`.

```
N = CoordSys3D('N')
x0, x1, x2 = var('x0, x1, x2')
vt = Vector.zero + x0*N.i + x1*N.j + x2*N.k
normt = vt.dot(vt) - 1
normt
```

which produces the first equation

```
x0**2 + x1**2 + x2**2 - 1
```

The second equation is

```
x0*x3 + x1*x4 + x2*x5
```

is computed by the code

```
x3, x4, x5 = var('x3, x4, x5')
vm = Vector.zero + x3*N.i + x4*N.j + x5*N.k
momvt = vt.dot(vm)
```

The radii are `[0.5, 0.5, 0.5, 0.5]` defined by

```
radii = [radius for _ in range(4)]
```

The polynomial system is constructed by

```
eqs = [normt, momvt]
for (ctr, rad) in zip(centers, radii):
    vc = Vector.zero + ctr[0]*N.i + ctr[1]*N.j + ctr[2]*N.k
    left = vm - vc.cross(vt)
    equ = left.dot(left) - rad**2
    eqs.append(equ)
```

To apply the blackbox solver, we have to convert the polynomials to strings.

```
fourspheres = []
print('the polynomial system :')
for pol in eqs:
    print(pol)
    fourspheres.append(str(pol) + ';')
```

The output to the above code cell is

```
the polynomial system :
x0**2 + x1**2 + x2**2 - 1
x0*x3 + x1*x4 + x2*x5
x3**2 + x4**2 + x5**2 - 0.25
x3**2 + (-x1 + x5)**2 + (x2 + x4)**2 - 0.25
(-0.866025403784439*x2 + x3)**2 + (0.5*x2 + x4)**2 + (0.866025403784439*x0 - 0.5*x1 +␣
↪x5)**2 - 0.25
(-0.816496580927726*x0 + 0.5*x2 + x4)**2 + (0.288675134594813*x0 - 0.5*x1 + x5)**2 + (0.
↪816496580927726*x1 - 0.288675134594813*x2 + x3)**2 - 0.25
```

So, we have six polynomial equations in six unknowns.

2.5.3 solving the problem

Now we call the blackbox solver.

```
sols = solve(fourspheres)

for (idx, sol) in enumerate(sols):
    print('Solution', idx+1, ':')
    print(sol)
```

The solution list is shown below:

```
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 4
the solution for t :
 x0 :  1.82013100766029E-16   2.92227989168779E-16
 x1 : -8.16496580927726E-01  -2.50326444773076E-17
 x2 : -5.77350269189626E-01   3.54015053218724E-17
 x3 :  6.04879596033482E-17   3.06586029409515E-17
 x4 :  2.88675134594813E-01  -1.77007526609362E-17
 x5 : -4.08248290463863E-01  -1.25163222386536E-17,
== err :  4.981E-16 = rco :  1.657E-17 = res :  3.821E-16 =
Solution 2 :
```

(continues on next page)

(continued from previous page)

```
t :  1.00000000000000E+00   0.00000000000000E+00
m : 4
the solution for t :
 x0 : -7.07106781186547E-01  -2.17839875856796E-16
 x1 : -4.08248290463863E-01   2.08243914343071E-16
 x2 :  5.77350269189626E-01  -1.19547586767375E-16
 x3 :  2.50000000000000E-01  -3.72860037233369E-31
 x4 : -4.33012701892219E-01   2.77333911991762E-31
 x5 : -1.99196604815539E-16   2.50510368981921E-16
== err :  1.118E-15 = rco :  2.133E-17 = res :  4.441E-16 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 4
the solution for t :
 x0 :  7.07106781186547E-01   1.37982017054626E-16
 x1 : -4.08248290463863E-01   3.39894708038934E-18
 x2 :  5.77350269189626E-01  -1.66589349202482E-16
 x3 :  2.50000000000000E-01  -4.09837892165604E-31
 x4 : -1.44337567297407E-01   1.66589349202482E-16
 x5 : -4.08248290463863E-01  -5.88982292472640E-17
== err :  1.023E-15 = rco :  4.667E-17 = res :  3.331E-16 =
Solution 4 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 4
the solution for t :
 x0 : -7.07106781186547E-01   8.51323534940577E-17
 x1 :  4.08248290463863E-01  -9.82010876877884E-18
 x2 : -5.77350269189626E-01  -1.11209278833498E-16
 x3 : -2.50000000000000E-01   6.61657084254124E-29
 x4 :  1.44337567297406E-01   1.11209278833581E-16
 x5 :  4.08248290463863E-01  -3.93184175971020E-17
== err :  2.006E-14 = rco :  1.477E-17 = res :  5.551E-16 =
Solution 5 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 4
the solution for t :
 x0 :  7.07106781186548E-01   2.00971395228568E-16
 x1 :  4.08248290463863E-01  -2.45578336010813E-16
 x2 : -5.77350269189626E-01   7.24885788968468E-17
 x3 : -2.50000000000000E-01  -2.77333911991762E-31
 x4 :  4.33012701892219E-01  -3.82104500966428E-31
 x5 : -7.32462262249068E-17  -2.71206918859082E-16
== err :  7.741E-16 = rco :  4.417E-17 = res :  3.886E-16 =
Solution 6 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 4
the solution for t :
 x0 : -3.22358540809185E-16   5.45289082605370E-16
 x1 :  8.16496580927726E-01  -1.64014136987415E-17
 x2 :  5.77350269189625E-01   2.31951016948522E-17
 x3 : -1.74561355056418E-16   2.00875473111053E-17
 x4 : -2.88675134594813E-01  -1.15975508474262E-17
```

(continues on next page)

(continued from previous page)

```
 x5 :  4.08248290463863E-01  -8.20070684937081E-18
== err :  5.471E-16 = rco :  1.448E-17 = res :  3.682E-16 =
```

Observe the `m :  4` which indicates the multiplicy four of each solution.

2.5.4 the tangent lines

The solutions contain the components of the tangent and the moment vectors from which the tangent lines can be computed.

```
def tangent_lines(solpts, verbose=True):
    """
    Given in solpts is the list of solution points,
    the tuples which respresent the tangent lines
    are returned in a list.
    Each tuple contains a point on the line
    and the tangent vector.
    """
    result = []
    for point in solpts:
        if verbose:
            print(point, end='')
        tan = Vector.zero + point[0]*N.i + point[1]*N.j + point[2]*N.k
        mom = Vector.zero + point[3]*N.i + point[4]*N.j + point[5]*N.k
        pnt = tan.cross(mom) # solves m = p x t
        pntcrd = (pnt.dot(N.i), pnt.dot(N.j), pnt.dot(N.k))
        tancrd = (tan.dot(N.i), tan.dot(N.j), tan.dot(N.k))
        if verbose:
            print(', appending :', pntcrd)
        result.append((pntcrd, tancrd))
    return result
```

The input to the `tangent_lines` function is computed below:

```
crd = [coordinates(sol) for sol in sols]
complexpoints = [values for (names, values) in crd]
points = []
for point in complexpoints:
    vals = []
    for values in point:
        vals.append(values.real)
    points.append(tuple(vals))
```

and then the tangents are computed as

```
tangents = tangent_lines(points)
```

2.5.5 plotting the lines

Let us first plot the four spheres…

```
%matplotlib widget
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
u = np.linspace(0, 2 * np.pi, 100)
v = np.linspace(0, np.pi, 100)
R = float(radius)
x1 = float(ctr1[0]) + R * np.outer(np.cos(u), np.sin(v))
y1 = float(ctr1[1]) + R * np.outer(np.sin(u), np.sin(v))
z1 = float(ctr1[2]) + R * np.outer(np.ones(np.size(u)), np.cos(v))
x2 = float(ctr2[0]) + R * np.outer(np.cos(u), np.sin(v))
y2 = float(ctr2[1]) + R * np.outer(np.sin(u), np.sin(v))
z2 = float(ctr2[2]) + R * np.outer(np.ones(np.size(u)), np.cos(v))
x3 = float(ctr3[0]) + R * np.outer(np.cos(u), np.sin(v))
y3 = float(ctr3[1]) + R * np.outer(np.sin(u), np.sin(v))
z3 = float(ctr3[2]) + R * np.outer(np.ones(np.size(u)), np.cos(v))
x4 = float(ctr4[0]) + R * np.outer(np.cos(u), np.sin(v))
y4 = float(ctr4[1]) + R * np.outer(np.sin(u), np.sin(v))
z4 = float(ctr4[2]) + R * np.outer(np.ones(np.size(u)), np.cos(v))
# Plot the surfaces
sphere1 = ax.plot_surface(x1, y1, z1, alpha=0.8)
sphere2 = ax.plot_surface(x2, y2, z2, alpha=0.8)
sphere3 = ax.plot_surface(x3, y3, z3, alpha=0.8)
sphere3 = ax.plot_surface(x4, y4, z4, alpha=0.8)
# Set an equal aspect ratio
ax.set_aspect('equal')
plt.show()
```

The output of the code cell is in Fig. 2.12.

The second figure in Fig. 2.13 shows the tangent lines.

```
%matplotlib widget
ax = plt.figure().add_subplot(projection='3d')
# range of the tangent lines
theta = np.linspace(-2.5, 2.5, 10)
pnt1, tan1 = tangents[0]
x1 = float(pnt1[0]) + theta*tan1[0]
y1 = float(pnt1[1]) + theta*tan1[1]
z1 = float(pnt1[2]) + theta*tan1[2]
pnt2, tan2 = tangents[1]
x2 = float(pnt2[0]) + theta*tan2[0]
y2 = float(pnt2[1]) + theta*tan2[1]
z2 = float(pnt2[2]) + theta*tan2[2]
pnt3, tan3 = tangents[2]
x3 = float(pnt3[0]) + theta*tan3[0]
y3 = float(pnt3[1]) + theta*tan3[1]
z3 = float(pnt3[2]) + theta*tan3[2]
pnt4, tan4 = tangents[3]
x4 = float(pnt4[0]) + theta*tan4[0]
y4 = float(pnt4[1]) + theta*tan4[1]
```

(continues on next page)

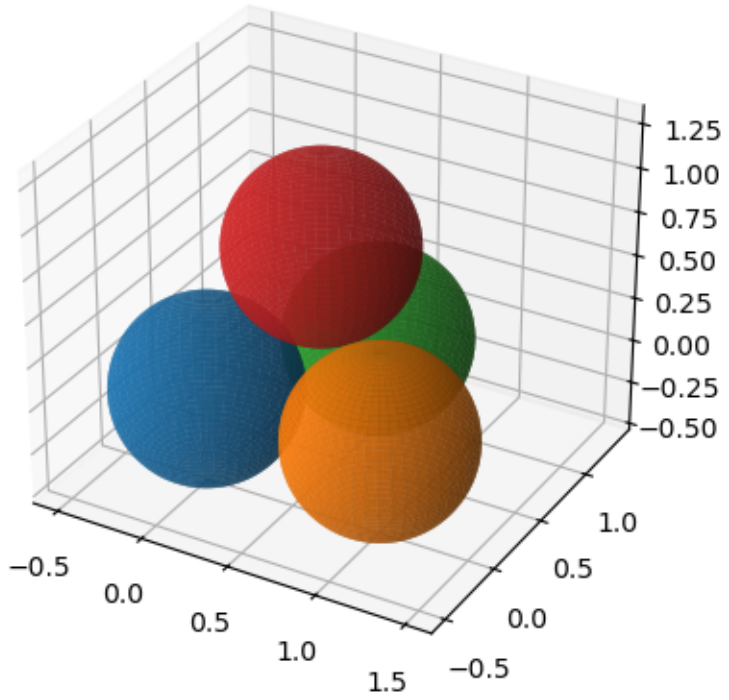


Fig. 2.12: Four touching spheres.

(continued from previous page)

```
z4 = float(pnt4[2]) + theta*tan4[2]
pnt5, tan5 = tangents[4]
x5 = float(pnt5[0]) + theta*tan5[0]
y5 = float(pnt5[1]) + theta*tan5[1]
z5 = float(pnt5[2]) + theta*tan5[2]
pnt6, tan6 = tangents[5]
x6 = float(pnt6[0]) + theta*tan6[0]
y6 = float(pnt6[1]) + theta*tan6[1]
z6 = float(pnt6[2]) + theta*tan6[2]
line1 = ax.plot(x1, y1, z1)
line2 = ax.plot(x2, y2, z2)
line3 = ax.plot(x3, y3, z3)
line4 = ax.plot(x4, y4, z4)
line5 = ax.plot(x5, y5, z5)
line6 = ax.plot(x6, y6, z6)
# Set an equal aspect ratio
ax.set_aspect('equal')
plt.show()
```

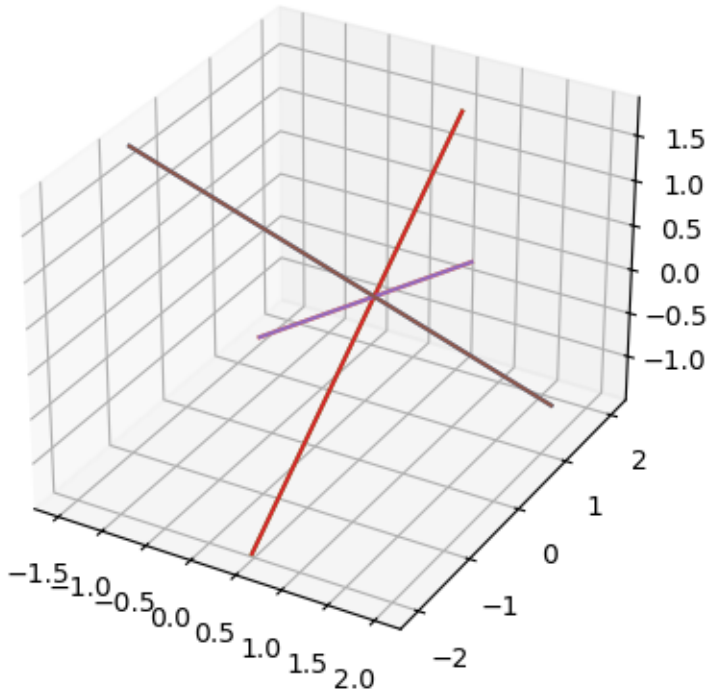


Fig. 2.13: The computed tangent lines.

And then we plot the spheres and the tangent lines:

```
%matplotlib widget
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
u = np.linspace(0, 2 * np.pi, 100)
v = np.linspace(0, np.pi, 100)
R = float(radius)
x1 = float(ctr1[0]) + R * np.outer(np.cos(u), np.sin(v))
y1 = float(ctr1[1]) + R * np.outer(np.sin(u), np.sin(v))
z1 = float(ctr1[2]) + R * np.outer(np.ones(np.size(u)), np.cos(v))
x2 = float(ctr2[0]) + R * np.outer(np.cos(u), np.sin(v))
y2 = float(ctr2[1]) + R * np.outer(np.sin(u), np.sin(v))
z2 = float(ctr2[2]) + R * np.outer(np.ones(np.size(u)), np.cos(v))
x3 = float(ctr3[0]) + R * np.outer(np.cos(u), np.sin(v))
y3 = float(ctr3[1]) + R * np.outer(np.sin(u), np.sin(v))
z3 = float(ctr3[2]) + R * np.outer(np.ones(np.size(u)), np.cos(v))
x4 = float(ctr4[0]) + R * np.outer(np.cos(u), np.sin(v))
y4 = float(ctr4[1]) + R * np.outer(np.sin(u), np.sin(v))
z4 = float(ctr4[2]) + R * np.outer(np.ones(np.size(u)), np.cos(v))
# Plot the surfaces
sphere1 = ax.plot_surface(x1, y1, z1, alpha=0.8)
sphere2 = ax.plot_surface(x2, y2, z2, alpha=0.8)
sphere3 = ax.plot_surface(x3, y3, z3, alpha=0.8)
sphere4 = ax.plot_surface(x4, y4, z4, alpha=0.8)
# range of the tangent lines
theta = np.linspace(-2.5, 2.5, 10)
pnt1, tan1 = tangents[0]
x1 = float(pnt1[0]) + theta*tan1[0]
y1 = float(pnt1[1]) + theta*tan1[1]
z1 = float(pnt1[2]) + theta*tan1[2
pnt2, tan2 = tangents[1]
x2 = float(pnt2[0]) + theta*tan2[0]
y2 = float(pnt2[1]) + theta*tan2[1]
z2 = float(pnt2[2]) + theta*tan2[2]
pnt3, tan3 = tangents[2]
x3 = float(pnt3[0]) + theta*tan3[0]
y3 = float(pnt3[1]) + theta*tan3[1]
z3 = float(pnt3[2]) + theta*tan3[2]
pnt4, tan4 = tangents[3]
x4 = float(pnt4[0]) + theta*tan4[0]
y4 = float(pnt4[1]) + theta*tan4[1]
z4 = float(pnt4[2]) + theta*tan4[2]
pnt5, tan5 = tangents[4]
x5 = float(pnt5[0]) + theta*tan5[0]
y5 = float(pnt5[1]) + theta*tan5[1]
z5 = float(pnt5[2]) + theta*tan5[2]
pnt6, tan6 = tangents[5]
x6 = float(pnt6[0]) + theta*tan6[0]
y6 = float(pnt6[1]) + theta*tan6[1]
z6 = float(pnt6[2]) + theta*tan6[2]
line1 = ax.plot(x1, y1, z1)
line2 = ax.plot(x2, y2, z2)
line3 = ax.plot(x3, y3, z3)
line4 = ax.plot(x4, y4, z4)
```

(continues on next page)

(continued from previous page)

```
line5 = ax.plot(x5, y5, z5)
line6 = ax.plot(x6, y6, z6)
# Set an equal aspect ratio
ax.axes.set_xlim3d(-1.5, 1.5)
ax.axes.set_ylim3d(-1.5, 1.5)
ax.axes.set_zlim3d(-1.5, 1.5)
ax.set_aspect('equal')
ax.view_init(elev=30, azim=30, roll=0)
plt.show()
```

which produces Fig. 2.14.

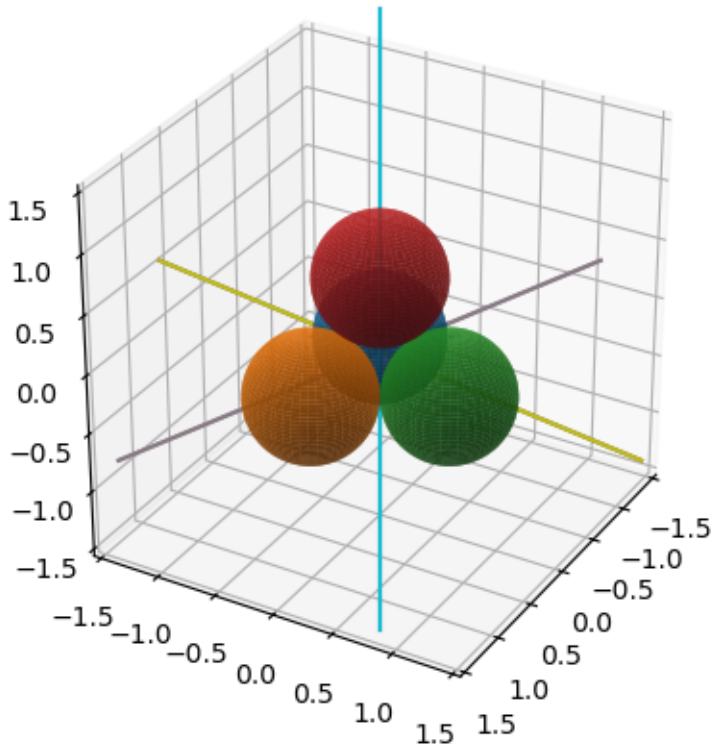


Fig. 2.14: All lines tangent to four spheres.

2.6 Tracking Paths Step by Step

One main benefit of phcpy is the interactivity and the inversion of control. Instead of the path tracker deciding the pace of computing points on the path, the user can ask for the next point on the path, which is convenient to plot the points on the path.

2.6.1 plotting solutions paths

In the example below, the real parts of the solution paths are plotted, in Fig. 2.15.

```
from phcpy.dimension import set_seed, get_seed
from phcpy.solutions import strsol2dict
from phcpy.starters import total_degree_start_system
from phcpy.trackers import initialize_double_tracker
from phcpy.trackers import initialize_double_solution
from phcpy.trackers import next_double_solution
```

The construction of the homotopy depends on the generation of random numbers. To obtain consistently the same plots, the seed of the random number generator is fixed.

```
set_seed(12871)
print('the seed :', get_seed())
```

What is printed is `the seed :  12871`.

The system that will be solved is defined as follows:

```
p = ['x^2 + y - 3;', 'x + 0.125*y^2 - 1.5;']
```

For this intersection of two quadrics, we construct a total degree start system and compute four start solutions:

```
q, qsols = total_degree_start_system(p)
for pol in q:
    print(pol)
```

Then here is the start system:

```
x^2 - 1;
y^2 - 1;
```

```
print('number of start solutions :', len(qsols))
```

and 4 is printed.

```
initialize_double_tracker(p, q, False)

plt.ion()
```

The code

```
fig = plt.figure()
for k in range(len(qsols)):
    if(k == 0):
```

(continues on next page)

(continued from previous page)

```
        axs = fig.add_subplot(221)
    elif(k == 1):
        axs = fig.add_subplot(222)
    elif(k == 2):
        axs = fig.add_subplot(223)
    elif(k == 3):
        axs = fig.add_subplot(224)
    startsol = qsols[k]
    initialize_double_solution(len(p),startsol)
    dictsol = strsol2dict(startsol)
    xpoints =  [dictsol['x']]
    ypoints =  [dictsol['y']]
    for k in range(300):
        ns = next_double_solution()
        dictsol = strsol2dict(ns)
        xpoints.append(dictsol['x'])
        ypoints.append(dictsol['y'])
        tval = dictsol['t'].real
        if(tval == 1.0):
            break
    print(ns)
    xre = [point.real for point in xpoints]
    yre = [point.real for point in ypoints]
    axs.set_xlim(min(xre)-0.3, max(xre)+0.3)
    axs.set_ylim(min(yre)-0.3, max(yre)+0.3)
    dots, = axs.plot(xre,yre,'r-')
    dots, = axs.plot(xre,yre,'ro')
    fig.canvas.draw()
fig.canvas.draw()
```

prints the solutions at the end of the paths

```
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.00000071115204E+00  -4.02767598012276E-06
 y :  1.99999857771171E+00   8.05535854116567E-06
= err :  2.614E-06 = rco :  1.000E+00 = res :  9.361E-13 =
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  9.99994847664516E-01  -1.64638311049799E-06
 y :  2.00001030465627E+00   3.29275198635159E-06
== err :  7.006E-06 = rco :  1.000E+00 = res :  1.187E-11 =
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  9.99999462998807E-01   3.01282709881038E-06
 y :  2.00000107400770E+00  -6.02565521719758E-06
== err :  6.245E-06 = rco :  1.000E+00 = res :  7.729E-12 =
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
```

(continues on next page)

(continued from previous page)

```
the solution for t :
 x : -3.00000000000000E+00   1.39180902310149E-15
 y : -6.00000000000001E+00   5.55716009286787E-15
== err :  4.185E-07 = rco :  1.000E+00 = res :  2.071E-14 =
```

and shows then the plot

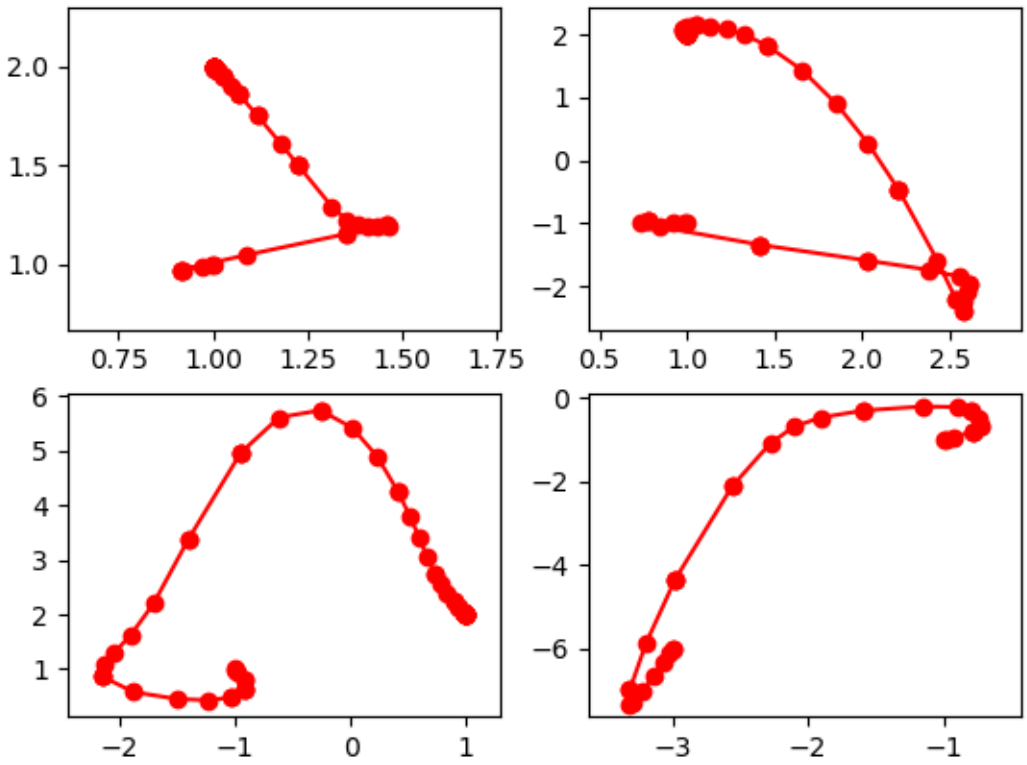


Fig. 2.15: The real parts of four solution paths.

Typically, with an adaptive step size control, the points are closer to each other at the start and end of the paths, and where the paths turn. The `trackers` module exports the original path trackers, which use *aposteriori step size control*. An aposteriori step size control algorithm determines the step size based on the performance of the corrector.

2.6.2 plotting paths and poles

The *apriori step size control* determines the step size based on the location of the nearest pole and the curvature. In addition to the (real parts of the paths), the location of the nearest poles is plotted.

```
import matplotlib.pyplot as plt

from phcpy.dimension import set_seed, get_seed
from phcpy.solutions import strsol2dict
from phcpy.starters import total_degree_start_system
from phcpy.curves import set_default_parameters, write_parameters
from phcpy.curves import initialize_double_artificial_homotopy
from phcpy.curves import set_double_solution, get_double_solution
from phcpy.curves import double_predict_correct
from phcpy.curves import double_t_value, double_closest_pole
```

The seed is fixed to obtain the same plots in each run.

```
set_seed(12871)
```

The system that will be solved is defined as

```
p = ['x^2 + y - 3;', 'x + 0.125*y^2 - 1.5;']
```

and a start system based on the total degree is constructed:

```
x^2 - 1;
y^2 - 1;
```

as the output of the code

```
q, qsols = total_degree_start_system(p)
for pol in q:
    print(pol)
```

The list *qsols* has the four solutions of the start system *q*.

Before lauching the path trackers, the parameters must be set. Default values are used

```
set_default_parameters()
write_parameters()
```

and shown below:

```
Values of the HOMOTOPY CONTINUATION PARAMETERS :
 1. gamma : (-0.8063005962200716-0.5915060004219376j)
 2. degree of numerator of Pade approximant    : 5
 3. degree of denominator of Pade approximant  : 1
 4. maximum step size                          : 0.1
 5. minimum step size                          : 1e-06
 6. multiplication factor for the pole radius  : 0.5
 7. multiplication factor for the curvature    : 0.005
 8. tolerance on the residual of the predictor : 0.001
 9. tolerance on the residual of the corrector : 1e-08
10. tolerance on zero series coefficients      : 1e-12
```

(continues on next page)

(continued from previous page)

```
11. maximum number of corrector steps          : 4
12. maximum steps on a path                    : 1000
```

Then the homotopy is constructed:

```
initialize_double_artificial_homotopy(p, q, False)
```

Then the code below

```
plt.ion()
fig1 = plt.figure()
allpoles = []
for k in range(len(qsols)):
    if(k == 0):
        axs = fig1.add_subplot(221)
    elif(k == 1):
        axs = fig1.add_subplot(222)
    elif(k == 2):
        axs = fig1.add_subplot(223)
    elif(k == 3):
        axs = fig1.add_subplot(224)
    startsol = qsols[k]
    set_double_solution(len(p), startsol)
    dictsol = strsol2dict(startsol)
    xpoints =  [dictsol['x']]
    ypoints =  [dictsol['y']]
    poles = []
    for k in range(100):
        ns = get_double_solution()
        dictsol = strsol2dict(ns)
        xpoints.append(dictsol['x'])
        ypoints.append(dictsol['y'])
        tval = dictsol['t'].real
        if(tval == 1.0):
            break
        double_predict_correct()
        pole = double_closest_pole()
        tval = double_t_value()
        locp = (tval+pole[0], pole[1])
        poles.append(locp)
    print(ns)
    xre = [point.real for point in xpoints]
    yre = [point.real for point in ypoints]
    axs.set_xlim(min(xre)-0.3, max(xre)+0.3)
    axs.set_ylim(min(yre)-0.3, max(yre)+0.3)
    dots, = axs.plot(xre,yre,'b-')
    dots, = axs.plot(xre,yre,'bo')
    fig1.canvas.draw()
    allpoles.append(poles)
fig1.canvas.draw()
```

prints the solutions at the end of the path:

```
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.00000458683462E+00   7.95321001653953E-06
 y :  1.99999082635867E+00  -1.59064683422953E-05
== err :  1.458E-05 = rco :  3.155E-12 = res :  1.529E-12 =
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  9.99999434968695E-01   9.78667863138325E-07
 y :  2.00000113004483E+00  -1.95736651949408E-06
== err :  1.658E-05 = rco :  6.664E-12 = res :  1.976E-12 =
t :  1.00000000000000E+00   0.00000000000000E+00\
m : 1
the solution for t :
 x :  1.00000600449478E+00   2.32224203692682E-07
 y :  1.99998799104108E+00  -4.64448075210226E-07
== err :  1.671E-05 = rco :  9.377E-13 = res :  3.579E-12 =
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -3.00000000000000E+00   0.00000000000000E+00
 y : -6.00000000000000E+00   0.00000000000000E+00
== err :  5.551E-16 = rco :  1.965E-01 = res :  0.000E+00 =
```

and produces the plot in Fig. 2.16.

The poles are shown in Fig. 2.17 plotted with the code in

```
fig2 = plt.figure()
for k in range(len(qsols)):
    if(k == 0):
        axs = fig2.add_subplot(221)
    elif(k == 1):
        axs = fig2.add_subplot(222)
    elif(k == 2):
        axs = fig2.add_subplot(223)
    elif(k == 3):
        axs = fig2.add_subplot(224)
    poles = allpoles[k]
    pl0 = [pole[0] for pole in poles]
    pl1 = [pole[1] for pole in poles]
    axs.set_xlim(-0.2, 1.2)
    axs.set_ylim(-0.5, 0.5)
    dots, = axs.plot(pl0,pl1,'r+')
    fig2.canvas.draw()
fig2.canvas.draw()
```

Observe that for this example, more poles are located closer to the middle and the end of the paths.

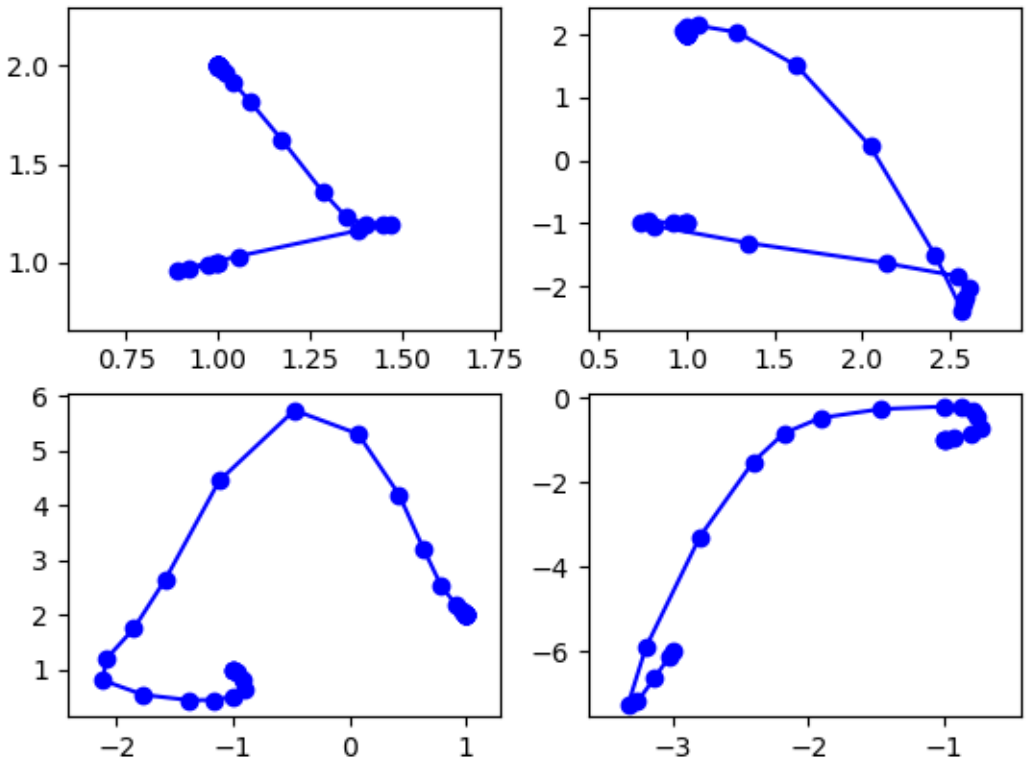


Fig. 2.16: The real parts of four solution paths.

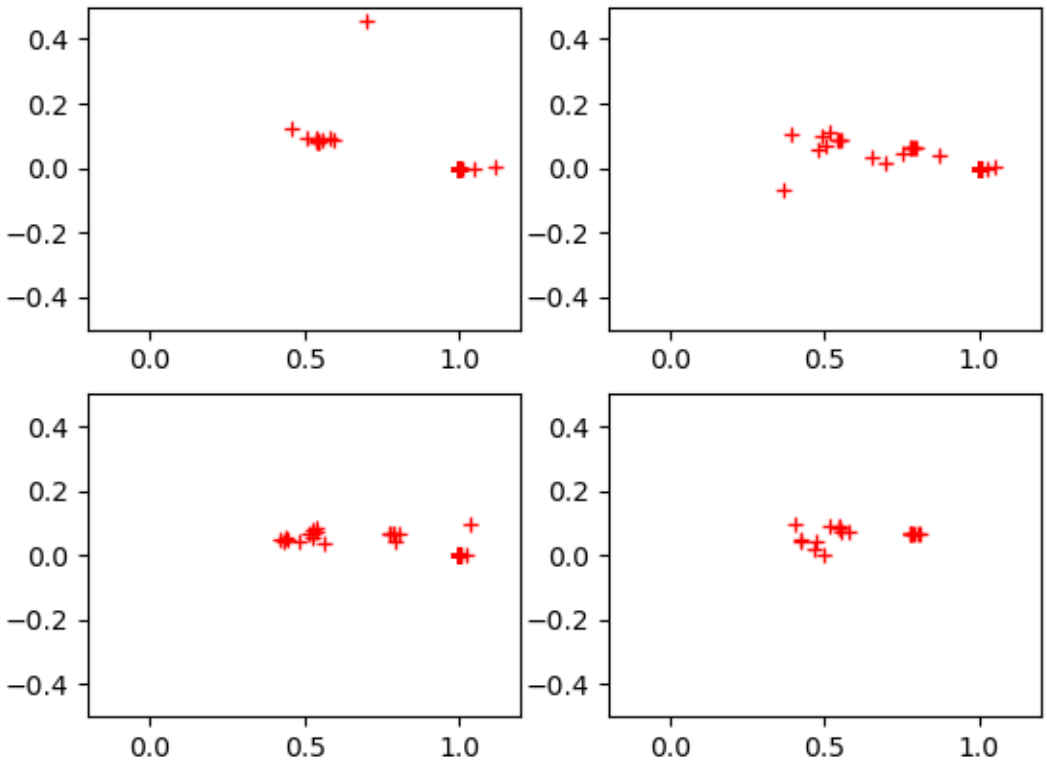


Fig. 2.17: The nearest poles of four solution paths.

2.7 Adjacent 2-by-2 Minors

This section documents the computation of the irreducible factors of a pure dimensional solution set, using the example of the 2-by-2 adjacent minors of a general 2-by-n matrix. The 2-by-2 minors of a 2-by-n matrix of indeterminates originates in algebraic statistics.

```
from phcpy.families import adjacent_minors
help(adjacent_minors)
```

shows

```
adjacent_minors(rows, cols)
    Returns all adjacent 2-by-2 minors of a general
    matrix of dimensions rows by cols.
    This system originated in a paper on lattice walks and
    primary decomposition, written by P. Diaconis, D. Eisenbud,
    and B. Sturmfels, published by Birkhauser in 1998 in
    Mathematical Essays in Honor of Gian-Carlo Rota,
    edited by B. E. Sagan and R. P. Stanley,
    volume 161 of Progress in Mathematics, pages 173--193.
    See also the paper by S. Hosten and J. Shapiro on
    Primary decomposition of lattice basis ideals, published in 2000
    in the Journal of Symbolic Computation, volume 29, pages 625-639.
```

Let us look at the simplest nontrivial case: the adjacent 2-by-2 minors of a 2-by-3 matrix.

```
pols = adjacent_minors(2, 3)
for pol in pols:
    print(pol)
```

shows

```
x_1_1*x_2_2-x_2_1*x_1_2;
x_1_2*x_2_3-x_2_2*x_1_3;
```

We have two polynomials in six variables. Therefore, we expect the solution set to be four dimensional. The two polynomials are quadrics. So, the degree of the solution set is expected to be four. The question is whether the four dimensional solution set of degree four is irreducible or not.

2.7.1 computing a witness set

A *witness set* of a positive dimensional solution set consists of

1. the original polynomial system, augmented with as many linear equations as the dimension of the solution set; and
2. generic points, as many as the degree of the solution set, computed as solutions of the augmented polynomial system.

The *embedding* adds extra slack variables, which are zero at the generic points.

The witness set data structure reduces the computation of a positive dimensional solution set to computing the isolated solutions of one embedded polynomial system.

```
from phcpy.sets import double_embed
epols = double_embed(6, 4, pols)
for pol in epols:
    print(pol)
```

shows

```
+ x_1_1*x_2_2 - x_2_1*x_1_2 + (-9.99086911101846E-01-4.27240455127090E-02*i)*zz1 + (4.
↪14818420957611E-01-9.09904213439104E-01*i)*zz2 + (-8.98599566975477E-01 + 4.
↪38769664210604E-01*i)*zz3 + (-9.87637111749394E-01 + 1.56757569180295E-01*i)*zz4;
- x_2_2*x_1_3 + x_1_2*x_2_3 + (-9.50915060423189E-01 + 3.09452012209265E-01*i)*zz1 + (8.
↪94934777859038E-01-4.46196978226426E-01*i)*zz2 + (4.28909177508030E-01-9.
↪03347617171477E-01*i)*zz3 + (9.99997394966914E-01 + 2.28255545098161E-03*i)*zz4;
zz1;
zz2;
zz3;
zz4;
+ (2.52642772862563E-01-1.64562207779935E-01*i)*x_1_1 + (2.38172268997908E-01-1.
↪84886617118381E-01*i)*x_2_2 + (-1.91482090565462E-01-2.32902769201594E-01*i)*x_2_1 +␣
↪(7.30954525688645E-02 + 2.92516915276440E-01*i)*x_1_2 + (-3.84959957593650E-02-2.
↪99043724594892E-01*i)*x_2_3 + (-2.71276842664000E-01-1.31597741406691E-01*i)*x_1_3 + (-
↪2.64408104251273E-04-3.01511228642393E-01*i)*zz1 + (2.72300014364985E-01 + 1.
↪29467343704581E-01*i)*zz2 + (-2.99939451765406E-01-3.07476207820803E-02*i)*zz3 + (-2.
↪95299363009813E-01 + 6.08882346195858E-02*i)*zz4 - 3.01511344577764E-01;
+ (-4.58249807046168E-01-5.40906757392526E-02*i)*x_1_1 + (4.14862726733922E-02 + 4.
↪11506360074024E-01*i)*x_2_2 + (-2.32018034572266E-01 + 9.48639573945220E-02*i)*x_2_1 +␣
↪(-2.80184386911322E-01-4.28818448828079E-01*i)*x_1_2 + (-1.20422832345040E-01-1.
↪42772536416847E-01*i)*x_2_3 + (-1.75000947987830E-01-2.52077630538672E-02*i)*x_1_3 +␣
↪(1.00853655248805E-01-1.59726738098879E-01*i)*zz1 + (1.47157089788836E-01 + 9.
↪31710575065957E-02*i)*zz2 + (-3.18937022210321E-02-2.67242905524245E-01*i)*zz3 + (-1.
↪70655540108348E-01 + 3.05332146451031E-02*i)*zz4+(-9.36346290234469E-02 + 2.
↪17662695080044E-01*i);
+ (-8.19176308116668E-02 + 9.31779891695395E-02*i)*x_1_1 + (-2.23021019174348E-01 + 9.
↪08742650844521E-02*i)*x_2_2 + (-2.55417845632647E-01 + 1.92893995983262E-01*i)*x_2_1 +␣
↪(2.85202344419800E-02 + 4.06697473728353E-01*i)*x_1_2 + (-3.26980478298163E-01 + 5.
↪39551406506719E-02*i)*x_2_3 + (-4.72487633926490E-03 + 3.03185406062837E-01*i)*x_1_3 +␣
↪(2.49469923033290E-01 + 7.22359362814556E-02*i)*zz1 + (3.57054498446798E-01 + 1.
↪84302643405729E-01*i)*zz2 + (2.44627852456931E-01-2.19409644826369E-02*i)*zz3 + (-6.
↪45211959942027E-02-2.81391298463502E-01*i)*zz4+(1.78511770624327E-02-2.88585493131170E-
↪01*i);
+ (-2.47934564140415E-01 + 3.78474051361417E-01*i)*x_1_1 + (2.57696009847760E-01-1.
↪97353959339253E-01*i)*x_2_2 + (-1.32713498879035E-01-6.68083554253052E-02*i)*x_2_1 +␣
↪(1.71277270038441E-01-1.69671423207279E-01*i)*x_1_2 + (2.45498609773496E-01-3.
↪00565440004147E-01*i)*x_2_3 + (-1.33200278933953E-01 + 9.57961946142634E-02*i)*x_1_3 +␣
↪(2.53856038005847E-01 + 2.28882119288700E-01*i)*zz1 + (3.08364764282092E-03-3.
↪24002619621010E-01*i)*zz2 + (2.00774472927742E-01 + 2.60472818035180E-01*i)*zz3 + (-2.
↪51374429242122E-01 + 8.28490424271253E-03*i)*zz4+(-7.60100938321149E-02-1.
↪82187404809906E-01*i);
```

Now we compute the second part of the witness set, using the blackbox solver.

```
from phcpy.solver import solve
esols = solve(epols)
```

(continues on next page)

(continued from previous page)

```
for (idx, sol) in enumerate(esols):
    print('Solution', idx+1, ':')
    print(sol)
```

shows four generic points on the four dimensional solution set:

```
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x_1_1 :  1.26098995657825E+00   1.23960253907080E-01
 x_2_2 :  6.08716792661783E-02  -3.72692628829057E-01
 x_2_1 : -1.49366498111755E+00  -1.72487935488432E+00
 x_1_2 :  1.17926528430272E-01   1.73403649424959E-01
 zz1 :  2.01661798954576E-33  -1.36621089687380E-31
 zz2 :  4.18934314423693E-32   1.19013821639201E-31
 zz3 :  9.09330358138888E-33  -3.97471598577921E-32
 zz4 :  9.41147475091108E-32  -5.25062461176711E-32
 x_1_3 : -5.43007038294243E-01   4.77552159454532E-01
 x_2_3 :  1.30126856409899E+00   4.91781165629461E-02
== err :  4.606E-15 = rco :  1.659E-02 = res :  1.221E-15 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x_1_1 :  7.84232277025150E-01   2.84185732210647E-02
 x_2_2 :  1.09164662176987E-01  -6.94770373095709E-01
 x_2_1 :  3.30315020994409E-02  -9.45083305517748E-01
 x_1_2 :  5.76431528154133E-01   9.13299754350158E-02
 zz1 : -1.29868775020467E-32  -1.23245957744023E-32
 zz2 : -1.02927640426824E-32  -1.56443669914841E-32
 zz3 : -1.96403692130211E-32  -7.02597925370710E-33
 zz4 :  0.00000000000000E+00   0.00000000000000E+00
 x_1_3 : -6.32506552290242E-01   1.49836349424459E-01
 x_2_3 :  1.81539704759152E-01   7.61970172619267E-01
== err :  1.716E-15 = rco :  2.536E-02 = res :  1.499E-15 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x_1_1 : -1.07029652224429E+00  -2.84414365427269E-02
 x_2_2 :  4.26584686826893E-01   4.61346027545957E-01
 x_2_1 : -8.74524024788936E-02  -7.16513058512809E-01
 x_1_2 :  7.70137841287209E-01  -5.24903704207380E-01
 zz1 : -2.60172998762874E-31  -7.95558581219627E-32
 zz2 :  1.94241870643223E-31  -7.60476764044638E-32
 zz3 : -1.18161218057147E-31   6.47719980338975E-32
 zz4 :  3.21269292411328E-31   9.43964867510126E-32
 x_1_3 :  1.88771487997930E+00   2.52881775952061E+00
 x_2_3 : -1.49855102312210E+00   1.51018382382141E+00
== err :  5.217E-15 = rco :  1.384E-02 = res :  1.749E-15 =
Solution 4 :
```

(continues on next page)

(continued from previous page)

```
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x_1_1 :  1.03149817185821E+00   1.48840565152077E-01
 x_2_2 : -1.97069785649029E-31  -4.19546322472859E-32
 x_2_1 : -1.96423581053286E+00  -2.24826401526688E+00
 x_1_2 : -6.05197132478501E-32   6.74819240986552E-32
 zz1 : -4.35083051092473E-32   8.73329103580651E-32
 zz2 :  6.61155389590362E-32   7.61859548953403E-32
 zz3 :  2.10093447239302E-32  -1.28492740537369E-32
 zz4 :  0.00000000000000E+00   0.00000000000000E+00
 x_1_3 : -1.40341064895811E-01   1.23712580175709E+00
 x_2_3 :  1.45874399778069E+00   6.42372628489355E-02\
== err :  4.557E-15 = rco :  1.108E-02 = res :  1.332E-15 =
```

As expected, we find four solutions, equal to the degree of the solution set.

2.7.2 monodromy breakup

The *numerical irreducible decomposition* of a pure dimensional solution set is a list of tuples, each tuple represents an irreducible component, with two elements

1. the list of labels to the generic points in the witness set; and
2. the certificate of the linear trace.

The *monodromy breakup* algorithm refines the witness set, partitioning the generic points in the witness set corresponding to the irreducible components. The stop test in the monodromy looping algorithm is provided by the linear trace, which serves as a certificate for the numerical computations.

The breakup algorithm runs in verbose mode:

```
from phcpy.factor import double_monodromy_breakup
deco = double_monodromy_breakup(epols, esols, 4, verbose=True)
```

In verbose mode, the progress of the algorithm is printed:

```
... running monodromy loops in double precision ...
... initializing the grid for the linear trace ...
The diagnostics of the trace grid :
  largest error on the samples : 1.056863766729658e-14
  smallest distance between the samples : 1.1622396157009902
... starting loop 1 ...
new permutation : [2, 1, 3, 4]
number of factors : 4 -> 3
the decomposition :
  factor 1 : ([1, 2], 0.2775442892800747)
  factor 2 : ([3], 0.2775442892800704)
  factor 3 : ([4], 9.992007221626409e-16)
the permutation :  2 1 3 4 : 4 -> 3
calculated sum at samples : -6.39839733359069E-02   1.09505498482705E-01
value at the linear trace :  1.07936961323818E-02  -9.32611213290812E-02
calculated sum at samples : -1.22710696579410E-01   2.89111792077992E-02
```

(continues on next page)

(continued from previous page)

```
value at the linear trace : -1.97488366047695E-01   2.31677799019584E-01
calculated sum at samples :  2.68414486121697E-01  -5.27451814780890E-01
value at the linear trace :  2.68414486121697E-01  -5.27451814780890E-01
Certifying with linear trace test...
calculated sum at samples : -6.39839733359069E-02   1.09505498482705E-01
value at the linear trace :  1.07936961323818E-02  -9.32611213290812E-02
The witness points 1 2 do not define a factor.
The factorization cannot be certified.
... starting loop 2 ...
new permutation : [1, 2, 3, 4]
number of factors : 3 -> 3
the decomposition :
  factor 1 : ([1, 2], 0.2775442892800747)
  factor 2 : ([3], 0.2775442892800704)
  factor 3 : ([4], 9.992007221626409e-16)
... starting loop 3 ...
the permutation :  1 2 3 4 : 3 -> 3
calculated sum at samples : -6.39839733359069E-02   1.09505498482705E-01
value at the linear trace :  1.07936961323818E-02  -9.32611213290812E-02
calculated sum at samples : -1.22710696579410E-01   2.89111792077992E-02
value at the linear trace : -1.97488366047695E-01   2.31677799019584E-01
calculated sum at samples :  2.68414486121697E-01  -5.27451814780890E-01
value at the linear trace :  2.68414486121697E-01  -5.27451814780890E-01
Certifying with linear trace test...
calculated sum at samples : -6.39839733359069E-02   1.09505498482705E-01
value at the linear trace :  1.07936961323818E-02  -9.32611213290812E-02
The witness points 1 2 do not define a factor.
The factorization cannot be certified.
new permutation : [2, 3, 1, 4]
number of factors : 3 -> 2
the decomposition :
  factor 1 : ([1, 2, 3], 4.163336342344337e-15)
  factor 2 : ([4], 9.992007221626409e-16)
the permutation :  2 3 1 4 : 3 -> 2
calculated sum at samples : -1.86694669915317E-01   1.38416677690504E-01
value at the linear trace : -1.86694669915313E-01   1.38416677690503E-01
calculated sum at samples :  2.68414486121697E-01  -5.27451814780890E-01
value at the linear trace :  2.68414486121697E-01  -5.27451814780890E-01
Certifying with linear trace test...
calculated sum at samples : -1.86694669915317E-01   1.38416677690504E-01
value at the linear trace : -1.86694669915313E-01   1.38416677690503E-01
The witness points 1 2 3 define a factor.
calculated sum at samples :  2.68414486121697E-01  -5.27451814780890E-01
value at the linear trace :  2.68414486121697E-01  -5.27451814780890E-01
The witness points 4 define a factor.
The factorization is certified.
calculated sum at samples : -1.86694669915317E-01   1.38416677690504E-01
value at the linear trace : -1.86694669915313E-01   1.38416677690503E-01
calculated sum at samples :  2.68414486121697E-01  -5.27451814780890E-01
value at the linear trace :  2.68414486121697E-01  -5.27451814780890E-01
```

As a summary, the contents of `deco` is written as

```
from phcpy.factor import write_decomposition
write_decomposition(deco)
```

which produces

```
factor 1 : ([1, 2, 3], 4.163336342344337e-15)
factor 2 : ([4], 9.992007221626409e-16)
```

There are two irreducible factors, one of degree three, and another of degree one. The floating-point certificates are close to machine precision.

2.8 Design of a moving 7-bar mechanism

Laurent polynomial systems are systems that have negative exponents. In this section, we consider a Laurent system with one irreducible component of degree three and six isolated points.

A reference for the general case is the paper by Carlo Innocenti: **Polynomial solution to the position analysis of the 7-line Assur kinematic chain with one quaternary link**, in *Mech. Mach. Theory*, Vol. 30, No. 8, pages 1295-1303, 1995.

The special case was introduced in the paper with title: **Numerical decomposition of the solution sets of polynomial systems into irreducible components**, *SIAM J. Numer. Anal.* 38(6):2022-2046, 2001, by Andrew Sommese, Jan Verschelde, and Charles Wampler.

This special sevenbar mechanism has 6 isolated solutions and a cubic curve."

SymPy is used to define the equations, with complex arithmetic.

```
from cmath import exp
from sympy import var
```

From phcpy, the following functions are imported:

```
from phcpy.solutions import coordinates, diagnostics, condition_tables
from phcpy.solver import solve
from phcpy.sets import double_laurent_membertest
from phcpy.cascades import double_laurent_top_cascade
from phcpy.cascades import double_laurent_cascade_filter
from phcpy.factor import double_monodromy_breakup
```

2.8.1 a Laurent polynomial system

The code in this section defines the Laurent polynomial system for a generic instance of the parameters.

```
def symbolic_equations():
    """
    Returns the symbolic equations,
    with parameters a1, a2, a3, a4, a5, a6
    b0, b2, b3, b4, b5, and c0, with variables
    t1, t2, t3, t4, and t5.
    """
    a0, a1, a2, a3, a4, a5, a6 = var('a0, a1, a2, a3, a4, a5, a6')
```

(continues on next page)

(continued from previous page)

```
    b0, b2, b3, b4, b5, c0 = var('b0, b2, b3, b4, b5, c0')
    t1, t2, t3, t4, t5, t6 = var('t1, t2, t3, t4, t5, t6')
    eq1 = a1*t1 + a2*t2 - a3*t3 - a0
    eq2 = b2*t2 + a3*t3 - a4*t4 + a5*t5 - b0
    eq3 = a4*t4 + b5*t5 - a6*t6 - c0
    return [eq1, eq2, eq3]
```

Then the symbolic equations are computed via

```
eqs = symbolic_equations()
for equ in eqs:
    print(equ)
```

with output in

```
-a0 + a1*t1 + a2*t2 - a3*t3
a3*t3 - a4*t4 + a5*t5 - b0 + b2*t2
a4*t4 - a6*t6 + b5*t5 - c0
```

A generic instance of the problem is defined in the following function:

```
def generic_problem(eqs):
    """
    Given the symbolic equations in eqs,
    defines the equations for a generic problem,
    as a Laurent polynomial system.
    The system is returned as a list of string representations,
    suitable for input to the solve of phcpy.
    """
    i = complex(0, 1)
    subdict = {a0: 0.7 + 0.2*i, b0: 0.6, c0: 0.5 - 0.5*i, \
        a1: 0.7, a2: 0.8, b2: 0.6 + 0.5*i, a3: 0.4, a4: 0.6, \
        a5: 0.8, b5: 0.4 + 0.3*i, a6: 0.9}
    print(subdict)
    conjugates = {a0: 0.7 - 0.2*i, b0: 0.6, c0: 0.5 + 0.5*i, \
        a1: 0.7, a2: 0.8, b2: 0.6 - 0.5*i, a3: 0.4, a4: 0.6, \
        a5: 0.8, b5: 0.4 - 0.3*i, a6: 0.9}
    print(conjugates)
    result = []
    for equ in eqs:
        pol = equ.subs(subdict)
        result.append(str(pol) + ';')
    for equ in eqs:
        pol = str(equ.subs(conjugates))
        pol = pol.replace('t1', 't1**(-1)')
        pol = pol.replace('t2', 't2**(-1)')
        pol = pol.replace('t3', 't3**(-1)')
        pol = pol.replace('t4', 't4**(-1)')
        pol = pol.replace('t5', 't5**(-1)')
        pol = pol.replace('t6', 't6**(-1)')
        result.append(pol + ';')
    return result
```

Then the system is constructed symbolically via

```
T1, T2, T3, T4, T5, T6 = var('T1, T2, T3, T4, T5, T6')
generic = generic_problem(eqs)
for equ in generic:
    print(equ)
```

with output

```
0.7*t1 + 0.8*t2 - 0.4*t3 - 0.7 - 0.2*I;
t2*(0.6 + 0.5*I) + 0.4*t3 - 0.6*t4 + 0.8*t5 - 0.6;
0.6*t4 + t5*(0.4 + 0.3*I) - 0.9*t6 - 0.5 + 0.5*I;
0.7*t1**(-1) + 0.8*t2**(-1) - 0.4*t3**(-1) - 0.7 + 0.2*I;
t2**(-1)*(0.6 - 0.5*I) + 0.4*t3**(-1) - 0.6*t4**(-1) + 0.8*t5**(-1) - 0.6;
0.6*t4**(-1) + t5**(-1)*(0.4 - 0.3*I) - 0.9*t6**(-1) - 0.5 - 0.5*I;
```

Observe the negative exponents. Now, let us call the blackbox solver:

```
sols = solve(generic)
print('found', len(sols), 'solutions')
```

which prints `found 18 solutions`.

A condition table is a frequency table of the `err`, `rco`, and `res` fields of the solutions.

```
condition_tables(sols)
```

with output in

```
([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 16, 2],
 [3, 15, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 11, 7])
```

The first and third row indicate that the forward and backward errors of the solutions were all very small. The second row indicates that the estimates for the condition numbers were also in the good range. So, 18 solutions are well conditioned.

2.8.2 a special problem

As we have as many equations as variables, for general coefficients, we will have only isolated solutions. For special parameters, the system has an irreducible cubic as a solution set.

```
def special_parameters():
    """
    Returns a dictionary with special values for the parameters
    for the Assur7c in Roberts Cognate pattern.
    Before calling this function, the symbolic_equations()
    must have defined the variables for the parameters.
    """
    i = complex(0, 1)
    # start with the independent parameters
    result = {b0: 0.0, c0: 1.2, a2: 0.46, \
        b2: -0.11 + 0.49*i, a5: 0.41}
    theta4 = 0.6 + 0.8*i
```

(continues on next page)

(continued from previous page)

```
    theta3 = exp(1.8*i)
    # add the derived parameters
    result[a3] = result[a5]
    beta = result[b2]/result[a2]
    result[a0] = result[c0]/beta
    result[b5] = result[a5]*beta
    result[a4] = abs(result[b2])
    result[a1] = abs(result[a0] + result[a3]*theta3 - result[a4]*theta4/beta)
    result[a6] = abs(result[a4]*theta4 - result[b5]*theta3-result[c0])
    return result
```

```
def conjugates(dic):
    """
    Given on input a dictionary with variables as keys
    and complex numbers as values.
    Returns a dictionary with the same keys,
    but with values replaced by complex conjugates.
    """
    result = {}
    for key in list(dic.keys()):
        result[key] = dic[key].conjugate()
    return result
```

```
def special_problem(eqs):
    """
    Given the symbolic equations in eqs,
    replaces the parameters with special values.
    """
    pars = special_parameters()
    conj = conjugates(pars)
    result = []
    for equ in eqs:
        pol = equ.subs(pars)
        result.append(str(pol) + ';')
    for equ in eqs:
        pol = str(equ.subs(conj))
        pol = pol.replace('t1', 't1**(-1)')
        pol = pol.replace('t2', 't2**(-1)')
        pol = pol.replace('t3', 't3**(-1)')
        pol = pol.replace('t4', 't4**(-1)')
        pol = pol.replace('t5', 't5**(-1)')
        pol = pol.replace('t6', 't6**(-1)')
        result.append(pol + ';')
    return result
```

Constructing the polynomials of the special problem

```
special = special_problem(eqs)
for equ in special:
    print(equ)
```

leads to

```
0.710358341606049*t1 + 0.46*t2 - 0.41*t3 + 0.240761300555115 + 1.07248215701824*I;
t2*(-0.11 + 0.49*I) + 0.41*t3 - 0.502195181179589*t4 + 0.41*t5;
0.502195181179589*t4 + t5*(-0.0980434782608696 + 0.436739130434783*I) - 0.
↪775518556663656*t6 - 1.2;
0.710358341606049*t1**(-1) + 0.46*t2**(-1) - 0.41*t3**(-1) + 0.240761300555115 - 1.
↪07248215701824*I;
t2**(-1)*(-0.11 - 0.49*I) + 0.41*t3**(-1) - 0.502195181179589*t4**(-1) + 0.41*t5**(-1);
0.502195181179589*t4**(-1) + t5**(-1)*(-0.0980434782608696 - 0.436739130434783*I) - 0.
↪775518556663656*t6**(-1) - 1.2;
```

Running the `solve` of the `solver` module and printing the number of solutions

```
sols = solve(special)
print('found', len(sols), 'solutions')
```

shows `found 6 solutions`. Let us look at all solutions, executing

```
for (idx, sol) in enumerate(sols):
    print('Solution', idx+1, ':')
    print(sol)
```

which gives the output

```
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -5.66058532229597E-01  -3.67759063106022E-01
 t2 :  5.87958784214731E-01  -2.31024744989669E-01
 t3 :  2.66141319725173E-01   1.71943916132782E+00
 t4 :  2.42136902539446E-01   1.56435563278512E+00
 t5 : -8.79136932379914E-02  -5.67977370543055E-01
 t6 : -1.05957839449340E+00   1.03531112257767E+00
== err :  3.073E-15 = rco :  3.135E-02 = res :  9.853E-16 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  3.53717078322846E-01  -5.30879118400565E-01
 t2 : -5.99365365166588E-01  -1.57888836421937E+00
 t3 :  5.27607584716191E-01  -7.54168308853111E-02
 t4 :  5.86169848996842E-01  -8.37877878416834E-02
 t5 : -1.85739737768408E+00   2.65498503011424E-01
 t6 : -1.08247082572556E+00  -1.13383016827930E+00
== err :  2.542E-15 = rco :  3.053E-02 = res :  4.510E-16 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  8.69193907041793E-01  -1.30453665759482E+00
 t2 : -2.10146778358548E-01  -5.53582709999063E-01
 t3 :  1.85739737768408E+00  -2.65498503011422E-01
 t4 :  1.67183103323242E+00  -2.38973437749056E-01
```

(continues on next page)

(continued from previous page)

```
 t5 : -5.27607584716191E-01   7.54168308853116E-02
 t6 : -4.40509781239126E-01  -4.61410384022367E-01
== err :  8.299E-16 = rco :  2.577E-02 = res :  4.302E-16 =
Solution 4 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -5.45421549881402E-01  -8.38161877518281E-01
 t2 : -5.49340801920855E-01  -8.35598398361888E-01
 t3 : -9.74098087752273E-01   2.26125884049935E-01
 t4 :  1.78912046655936E-01  -9.83865071827120E-01
 t5 :  4.72153176961024E-02  -9.98884734979395E-01
 t6 : -8.74935009483375E-01  -4.84240363022669E-01
== err :  2.024E-15 = rco :  1.693E-01 = res :  6.661E-16 =
Solution 5 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -1.24225818334613E+00  -8.07075027813303E-01
 t2 :  1.47332994921904E+00  -5.78910775622768E-01
 t3 :  8.79136932379908E-02   5.67977370543054E-01
 t4 :  9.66290808831226E-02   6.24284218493861E-01
 t5 : -2.66141319725174E-01  -1.71943916132781E+00
 t6 : -4.82817017275396E-01   4.71759173981634E-01
== err :  2.046E-15 = rco :  3.752E-02 = res :  9.992E-16 =
Solution 6 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  2.80836878017556E-01  -9.59755514673061E-01
 t2 : -9.99161738567148E-01   4.09367827689738E-02
 t3 : -4.72153176961015E-02   9.98884734979395E-01
 t4 :  9.35633636119240E-01  -3.52972660360954E-01
 t5 :  9.74098087752273E-01  -2.26125884049935E-01
 t6 : -9.37276397924231E-01   3.48587082225057E-01
== err :  1.949E-15 = rco :  1.566E-01 = res :  7.910E-16 =
```

The output of

```
condition_tables(sols)
```

is

```
([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 1],
 [2, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6])
```

The `solve` of the solver module misses the component, but finds all isolated solutions.

2.8.3 a numerical irreducible decomposition

A numerical irreducible decomposition for this system augments the system with a linear equation and adds one slack variable. A cascade of homotopies find generic points on all positive dimensional components of the solution set.

The cascade is wrapped in the following function

```
def embed_and_cascade(pols, topdim):
    """
    Computes and solves an embedding at top dimension topdim
    of the Laurent polynomials in pols, before running one
    step in the cascade homotopy.
    Returns the embedded system, the three generic points,
    and the filtered solutions at the end of the cascade.
    """
    (embpols, sols0, sols1) \
        = double_laurent_top_cascade(len(pols), topdim, pols, 1.0e-08)
    print('the top generic points :')
    for (idx, sol) in enumerate(sols0):
        print('Solution', idx+1, ':')
        print(sol)
    print('the nonsolutions :')
    for (idx, sol) in enumerate(sols1):
        print('Solution', idx+1, ':')
        print(sol)
    print('... running cascade step ...')
    (embdown, nsols1, sols2) = double_laurent_cascade_filter(1, embpols, \
        sols1, 1.0e-8)
    filtsols2 = []
    for (idx, sol) in enumerate(nsols1):
        err, rco, res = diagnostics(sol)
        if res < 1.0e-8 and rco > 1.0e-8:
            _, point = coordinates(sol)
            crdpt = []
            for pt in point:
                crdpt.append(pt.real)
                crdpt.append(pt.imag)
            onset = double_laurent_membertest(embpols, sols0, 1, crdpt)
            if not onset:
                filtsols2.append(sol)
    print('... after running the cascade ...')
    for (idx, sol) in enumerate(filtsols2):
        print('Solution', idx+1, ':')
        print(sol)
    print('found %d isolated solutions' % len(filtsols2))
    return (embpols, sols0, filtsols2)
```

Running the code in

```
(embpols, sols0, isosols) = embed_and_cascade(special, 1)
```

produces the following output:

```
the top generic points :
```

(continues on next page)

(continued from previous page)

```
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -1.38546903964616E-01  -6.90082116709782E-01
 t2 :  1.73335780614296E+00  -1.87923014523194E+00
 t3 :  2.29192181084370E+00  -6.88217799479036E-01
 zz1 :  6.81370071005700E-17   1.65228741665612E-16
 t4 :  1.45392357364500E+00   2.10288883797136E+00
 t5 : -2.29192181084370E+00   6.88217799479037E-01
 t6 : -7.03671420729004E-01  -1.59719770241003E-02
== err :  1.018E-14 = rco :  7.104E-03 = res :  1.619E-15 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  9.95322784254993E-01   2.68098302301308E-02
 t2 : -2.29577075969873E-01  -2.62541647600257E-01
 t3 :  2.05412606828065E+00   2.36770142844668E+00
 zz1 :  7.58249556746807E-16  -7.20127010307089E-16
 t4 :  3.06452334567059E-01  -1.66495396855090E-01
 t5 : -2.05412606828065E+00  -2.36770142844668E+00
 t6 :  2.44172639794733E-01  -9.65280235905487E-01
== err :  3.577E-15 = rco :  2.031E-02 = res :  1.998E-15 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -2.82166573126121E-01   3.94770824953965E-01
 t2 : -1.34882663123749E-01  -2.78088349356538E+00
 t3 : -5.29856181081608E-02   1.79767069529071E-01
 zz1 :  2.43319655939393E-15   2.43565365274284E-15
 t4 :  2.74289769478703E+00   4.77512904043170E-01
 t5 :  5.29856181081606E-02  -1.79767069529072E-01
 t6 :  3.23379011328058E-01   3.61784458285676E-01
== err :  6.752E-15 = rco :  2.149E-03 = res :  2.746E-15 =
the nonsolutions :
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  2.06608664978716E-01  -5.16879012501813E-01
 t2 : -2.63044555866456E+00  -3.32754563190078E+00
 t3 :  2.63517362859822E-02  -3.68638950244269E-01
 zz1 :  2.85704892320864E-01  -1.03310141444983E+00
 t4 :  3.69849125743338E+00  -3.24986222144375E-01
 t5 :  2.43555779349728E+00   2.20779169965979E+00
 t6 : -9.87691023055072E-01   2.23467268352943E+00
== err :  3.403E-15 = rco :  2.764E-02 = res :  1.561E-15 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
```

(continues on next page)

(continued from previous page)

```
the solution for t :
 t1 :  3.09054535971898E-01   3.45124987046649E-02
 t2 : -2.62571332002632E-01   3.96698883381851E-02
 t3 :  4.32699725536785E+00   5.00970765602759E+00
 zz1 :  2.88323234772462E-01  -1.68997834567399E+00
 t4 : -1.07803480589631E+00  -1.04944159716503E+00
 t5 : -1.51252934187501E+00  -6.41341577084612E+00
 t6 :  1.32322533488782E+00   1.47762805502765E+00
== err :  6.549E-15 = rco :  1.025E-02 = res :  3.109E-15 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -4.38810628559853E-02  -3.73466290987713E-01
 t2 :  1.41948412243664E+00   2.64381540059301E-01
 t3 : -1.51714321706050E-01   2.82809534841746E+00
 zz1 :  5.80561465793845E-01   7.55877838451024E-01
 t4 :  6.84618807253357E-01  -1.45281628643651E-01
 t5 :  2.97376151600431E-01  -2.76806627220378E+00
 t6 : -3.91051079512042E-01  -5.02441609168954E-01
== err :  3.533E-15 = rco :  1.180E-02 = res :  1.086E-15 =
Solution 4 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -5.56053317223021E-01  -1.10476501709143E+00
 t2 :  1.08198361813419E+00  -6.10107625522898E-01
 t3 :  1.63797645826712E+00   2.65500692732987E-01
 zz1 : -3.75750870070081E-02  -3.41463274433182E-01
 t4 :  1.13327119828200E+00   1.00882972129605E+00
 t5 :  8.79031826440765E-02  -8.01012123191631E-01
 t6 : -2.97523163734206E-01   1.24044339222691E+00
== err :  1.312E-15 = rco :  2.758E-02 = res :  1.305E-15 =
Solution 5 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  1.42858931075131E-01   4.57096616580137E-01
 t2 : -1.11064648867271E-01  -1.72825737676934E-01
 t3 :  2.13308888441550E+00   1.81873215330586E+00
 zz1 : -7.57293747789502E-01  -3.06703567333995E-01
 t4 :  2.49672678368341E-01  -1.36328639570869E-01
 t5 : -1.84534929207840E+00  -3.88014927109987E+00
 t6 :  2.03214759107748E+00  -3.03532919113466E-01
== err :  7.406E-15 = rco :  6.149E-03 = res :  1.971E-15 =
Solution 6 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -4.75108750576191E-02  -1.75815265025158E-02
 t2 : -3.39314399121161E+00   1.41556569134256E+01
 t3 : -8.54021034392022E+00  -1.38811633117136E+01
```

(continues on next page)

(continued from previous page)

```
 zz1 : -1.13042166796985E+01   7.26160237898825E+00
 t4 :  4.31260891858428E-03  -2.20764606490238E-02
 t5 :  1.89710631587227E-02  -1.71967991631555E-02
 t6 :  1.24227998831989E+01  -1.02611559314054E+01
== err :  1.583E-14 = rco :  1.029E-05 = res :  1.698E-14 =
Solution 7 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  9.24280274976326E-01  -1.03328843987402E+00
 t2 : -5.37485466298478E-01  -6.93996812904627E-01
 t3 :  2.16870928120656E+00   3.05270120118574E-01
 zz1 :  1.72164678001449E-03  -2.61490044232207E-01
 t4 :  1.12688433550202E+00   6.04135856872397E-01
 t5 : -1.14709448190074E+00   7.21699657198301E-01
 t6 : -1.06009576375345E+00  -9.36416226240182E-03
== err :  1.235E-15 = rco :  1.803E-02 = res :  1.473E-15 =
Solution 8 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  2.56141977779414E-01  -1.26192262870549E-01
 t2 :  2.72348811313250E-02   2.27695964490337E+00
 t3 :  8.94179859360769E-02   1.64582692269313E-02
 zz1 : -1.69636690132142E+00   1.17291673807606E+00
 t4 :  1.44816411386502E-01  -5.21926135720004E-02
 t5 : -1.06047458297333E+00  -2.70699717824994E+00
 t6 :  2.29314581678167E+00  -1.93548691562723E+00
== err :  3.630E-15 = rco :  1.266E-03 = res :  2.873E-15 =
Solution 9 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -5.55849610226793E-01  -6.71303249412053E-01
 t2 :  2.22897995402302E+00  -6.36069101208733E-01
 t3 : -6.01231159130768E-03   3.20619531479791E-01
 zz1 :  1.91319173054801E-01   8.69593783238856E-01
 t4 :  1.46082467405615E+00   1.30115181744724E+00
 t5 : -2.81326700289881E-01  -5.36286320208369E-01
 t6 : -5.80370651666996E-01  -3.51674733202245E-01
== err :  1.098E-15 = rco :  2.893E-02 = res :  1.563E-15 =
Solution 10 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  1.96420789232812E-01  -7.25547806896222E-01
 t2 :  4.99261380871303E-01   6.57994832350889E-02
 t3 :  1.79119494447843E-01   8.39067157393641E-02
 zz1 : -2.92932332587968E-01   7.12592654282584E-01
 t4 :  1.31985073905489E+00   5.86875406583772E-01
 t5 : -2.16641539477234E-01  -1.59643525554850E-01
 t6 : -2.56051374441468E-01  -6.62533448829890E-01
```

(continues on next page)

(continued from previous page)

```
== err :  4.769E-16 = rco :  7.725E-03 = res :  1.402E-15 =
Solution 11 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -2.60846935289829E-01  -1.87501112823166E+00
 t2 :  8.91205034115317E-01   2.18216656676799E+00
 t3 :  1.35354427992550E-01   2.14130902473316E-01
 zz1 : -4.38444725918741E-01   6.37863637010385E-01
 t4 :  4.58998133742581E-01   3.40221617387902E-01
 t5 :  1.48618652575694E+00  -8.83671361618052E-01
 t6 : -4.27731643521526E-01   3.12634407365499E-01
== err :  4.740E-15 = rco :  1.451E-02 = res :  2.900E-15 =
Solution 12 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  2.58526368145855E-02  -5.89746935862251E-01
 t2 :  5.89197128653301E-01  -8.56471323438483E-01
 t3 :  4.99885121099350E-01  -8.51253650022305E-02
 zz1 : -1.40247285577845E-01   4.15696687647816E-01
 t4 :  1.66773190687860E+00   8.55179987286000E-01
 t5 : -3.91334700102683E-01   1.44828575079480E-01
 t6 : -3.52629174262130E-01  -2.31227315105538E-01
== err :  3.496E-15 = rco :  1.796E-02 = res :  1.877E-15 =
Solution 13 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  1.31675077902931E+00  -2.33335629490662E+00
 t2 :  2.60288716555775E-01  -6.40025748193045E-02
 t3 :  3.47791530603780E+00   3.39861528263726E+00
 zz1 :  1.78921624866824E+00  -9.20506290946835E-01
 t4 : -3.11182618390869E-02   1.27100852726915E-01
 t5 : -1.76684760979132E-01   1.89283790316579E-02
 t6 : -3.78392322972795E+00   1.30979364462129E+00
== err :  5.729E-15 = rco :  1.880E-03 = res :  3.067E-15 =
Solution 14 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  1.09751178036435E+00  -1.40574539820329E+00
 t2 : -6.26603734750229E-01   2.77315044013342E+00
 t3 :  1.16737341994583E-01   1.23560628499014E-01
 zz1 : -9.17943053708135E-01   1.14574950444327E+00
 t4 :  2.24794207999588E-01  -2.20864560734679E-01
 t5 :  7.19265347716089E-03  -2.96930745959293E-01
 t6 : -1.46858876699108E-01  -1.65019111957513E+00
== err :  9.467E-16 = rco :  1.061E-02 = res :  1.874E-15 =
Solution 15 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
```

(continues on next page)

(continued from previous page)

```
the solution for t :
 t1 : -1.30458602699707E+00  -2.33541493076135E+00
 t2 : -1.95199047692026E-01  -4.46316027103696E-01
 t3 : -2.63391619655473E-01  -2.31639399176850E-01
 zz1 :  3.72490200270314E-01  -8.90356341436546E-01
 t4 :  2.41781694569341E+00   1.09012175794204E-01
 t5 :  4.97584553326169E+00   7.63254192555898E-01
 t6 : -1.44788918406800E+00   3.95223854445634E+00
== err :  1.808E-15 = rco :  2.602E-02 = res :  1.971E-15 =
Solution 16 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  5.06717666964459E-01  -7.97241780193331E-01
 t2 : -4.84953123213085E-01  -4.42303335072244E-01
 t3 :  9.44820387296986E-01  -1.44512677763047E-02
 zz1 : -2.86872573899956E-01   1.14217012524121E-01
 t4 :  1.47330976885270E+00   3.99769298368997E-01
 t5 : -2.57072989456668E-01   3.67302107948625E-01
 t6 : -4.07703118258568E-01  -1.02532173760480E-01
== err :  1.760E-15 = rco :  2.381E-02 = res :  1.249E-15 =
Solution 17 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -4.44607365409264E-02  -2.59406017880835E-01
 t2 :  3.78176637095005E-01   4.39268560397966E-01
 t3 : -1.79503189927382E+00   3.20305158897225E+00
 zz1 :  6.51015867427647E-01   9.37169953082408E-01
 t4 :  1.30934173849756E+00  -1.12550388002783E+00
 t5 :  2.25665017724524E+00  -2.76677636166897E+00
 t6 : -3.40266607063703E-01  -2.61585700440509E-01
== err :  7.075E-15 = rco :  6.393E-03 = res :  2.526E-15 =
Solution 18 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -2.53488386324696E-01  -2.83321822997287E-01
 t2 :  4.84338946463415E-01  -1.67152891112537E+00
 t3 : -4.67320096443835E-01  -4.71307905321838E-02
 zz1 :  7.80393042398170E-02   4.84167415461090E-01
 t4 :  2.41580881604416E+00   6.84660143775337E-01
 t5 :  4.73818299916806E-01   3.62365346793219E-01
 t6 : -3.86544038004814E-01   4.76118474348958E-02
== err :  3.492E-15 = rco :  2.148E-02 = res :  1.291E-15 =
Solution 19 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -1.33188225629716E-01   2.10966347401895E-02
 t2 :  7.81125047936108E-02  -6.85635891220529E-02
 t3 : -1.17563370850181E+00  -1.23520999866901E+00
```

(continues on next page)

(continued from previous page)

```
 zz1 : -1.16759331681420E+00   1.23236695756379E+00
 t4 :  2.02453946582577E+00   5.47406959303102E-01
 t5 : -7.18144929758816E-02  -1.30397505946681E-01
 t6 :  1.24906033996779E+00  -1.34990480788646E+00
== err :  2.561E-15 = rco :  1.448E-03 = res :  5.407E-15 =
Solution 20 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  7.93825359960543E-01   3.42942332165835E-01
 t2 :  1.61651669536702E-01  -2.20420239291763E+00
 t3 :  4.64285871839968E+00   3.38285150308267E+00
 zz1 :  5.85868505889160E-01  -1.37232304716824E+00
 t4 :  4.40119366407873E-02   3.19449005533791E-01
 t5 : -3.57045518779285E+00  -3.30999449881425E+00
 t6 :  1.53654644664973E-01   4.28064658906619E-01
== err :  3.056E-15 = rco :  1.438E-02 = res :  3.844E-15 =
Solution 21 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -7.60489144456182E-02  -1.04300898291791E+00
 t2 :  1.30846500932422E+00   6.09095986219939E-02
 t3 :  2.60207018922791E+00   2.05366973304254E+00
 zz1 :  3.31325254376896E-01  -4.47605721252978E-01
 t4 :  5.02974219619557E-02   5.96655221299233E-01
 t5 : -8.46448240981595E-01  -2.38906321758213E+00
 t6 : -4.52524421602189E-01   8.14559254657865E-01
== err :  1.498E-15 = rco :  2.468E-02 = res :  2.047E-15 =
Solution 22 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  1.17748055149989E+00  -2.43215785209726E+00
 t2 : -1.11238676756403E+00   1.89832325651549E+00
 t3 :  3.71807527705904E+00   1.18513377244720E+00
 zz1 : -1.36994965965414E-01  -9.86163388302379E-01
 t4 : -5.27247356627358E-01  -7.14664957510453E-02
 t5 : -1.69492044434729E-01  -4.08782307130586E-01
 t6 : -1.38104781392739E+00   1.16797791507032E+00
== err :  2.038E-15 = rco :  4.306E-02 = res :  1.249E-15 =
Solution 23 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  3.12364985327808E+00  -2.05920731130273E+00
 t2 : -7.83955013846558E+00   3.44568114237919E+00
 t3 :  2.41311468318769E-01  -1.43824031559247E-02
 zz1 : -1.59787001054699E+00  -6.62918168050137E-01
 t4 :  1.10281361738592E+00  -3.47888278489006E+00
 t5 :  3.62179107231530E+00   1.85699344604169E+00
 t6 : -2.26905454742053E-01   2.75923139264321E-01
```

(continues on next page)

(continued from previous page)

```
== err :  1.042E-14 = rco :  4.252E-03 = res :  2.163E-15 =
Solution 24 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  3.08355119326816E+00  -8.60649488234224E+00
 t2 : -5.82366068292758E-01   6.35509246767222E+00
 t3 : -6.51342246123743E-02   9.06094708166825E-02
 zz1 :  2.84990305286854E+00   1.14816597303183E+00
 t4 :  4.34422945742045E-02   1.52135217854099E-01
 t5 :  6.74998864667608E+00   9.94704932815523E+00
 t6 : -1.17348229611637E+01   1.39527617933237E+00
== err :  3.194E-14 = rco :  5.571E-04 = res :  6.391E-15 =
Solution 25 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  3.99353498759161E-01  -1.05992514732995E+00
 t2 : -6.97576291634198E-01   2.90503809343657E-01
 t3 :  7.12853582076897E-02   4.30854721150634E-01
 zz1 : -1.84001525983760E-01   2.70200187419495E-01
 t4 :  1.46156773352004E+00  -1.50613908910889E-01
 t5 :  1.12282954533061E+00   4.33717117615447E-02
 t6 : -5.52341835801268E-01   1.66701841647516E-01
== err :  9.415E-16 = rco :  4.214E-02 = res :  7.841E-16 =
Solution 26 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  1.68976453558350E-01   4.67689996418420E-01
 t2 : -1.11289042160100E+00  -2.36087487730329E+00
 t3 :  7.14519021473384E-02   1.63214961074351E-01
 zz1 : -3.02872131405798E-01  -6.49699828630491E-02
 t4 :  2.59822631620836E+00   2.73920748230829E-02
 t5 : -5.72162586858151E-02  -1.87011959227602E-01
 t6 :  6.42736202016055E-01   6.82645862607672E-02
== err :  2.532E-15 = rco :  2.192E-03 = res :  2.207E-15 =
Solution 27 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  5.98088716113923E-01  -1.09928306831808E+00
 t2 : -1.32166665802307E+00   9.68674190067784E-02
 t3 :  1.25565072505728E+00   7.88279907441350E-01
 zz1 : -1.94309421837523E-01  -4.14018780046576E-01
 t4 :  1.15900115504862E+00  -3.40883097941626E-01
 t5 :  7.68268330674074E-01  -3.30650582363870E-01
 t6 : -4.24188159828689E-01   7.70802547523126E-01
== err :  1.539E-15 = rco :  3.512E-02 = res :  7.355E-16 =
Solution 28 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
```

(continues on next page)

(continued from previous page)

```
the solution for t :
" t1 :  1.74056293633403E+00   1.02810794797457E+00
 t2 : -4.53362942425264E+00  -3.21821306993922E+00
 t3 :  9.07980903275410E-01   2.06897987372685E+00
 zz1 :  9.08970112275608E-02  -1.10894123925719E+00
 t4 :  2.76795451024622E+00  -1.85436435929005E+00
 t5 :  8.31726772379002E-02  -3.06761087764399E-01
 t6 :  3.80053540200631E-01   3.19279779559526E-01
== err :  5.467E-15 = rco :  1.088E-02 = res :  4.621E-15 =
Solution 29 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  9.98996829187838E-02   5.45879605814945E+00
 t2 :  4.77713088756544E-02  -1.65064542093410E-01
 t3 :  7.03293595992006E-02  -5.75960210248865E-03
 zz1 : -4.34989172011554E+00   2.22547996626009E+00
 t4 :  7.42601541252893E-02  -4.06346066756681E-02
 t5 : -8.26936045369789E+00  -8.88179809231708E+00
 t6 :  9.96592014933793E+00  -6.77636361154879E+00
== err :  2.141E-14 = rco :  3.614E-04 = res :  4.914E-15 =
Solution 30 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  1.26911098400795E+00  -1.26566539297069E+00
 t2 : -8.06762092689544E-01   6.05358763159280E-01
 t3 :  4.36774346983282E+00   1.58729813776646E+00
 zz1 : -2.24457954433668E-01  -1.01428467390231E+00
 t4 : -5.06895216434008E-01   2.00062592651442E-01
 t5 : -2.24941445482227E+00  -1.41444572972009E+00
 t6 : -4.23738271342406E-01   3.28737704966059E-01
== err :  2.782E-15 = rco :  2.863E-02 = res :  1.360E-15 =
... running cascade step ...
Solution at position 14 is not appended.
Solution at position 22 is not appended.
... after running the cascade ...
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -5.45421549881402E-01  -8.38161877518281E-01
 t2 : -5.49340801920855E-01  -8.35598398361888E-01
 t3 : -9.74098087752273E-01   2.26125884049935E-01
 t6 : -8.74935009483375E-01  -4.84240363022669E-01
 t4 :  1.78912046655936E-01  -9.83865071827120E-01
 t5 :  4.72153176961023E-02  -9.98884734979395E-01
== err :  3.858E-16 = rco :  3.659E-02 = res :  1.388E-16 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
```

(continues on next page)

(continued from previous page)

```
 t1 : -5.66058532229596E-01  -3.67759063106022E-01
 t2 :  5.87958784214732E-01  -2.31024744989669E-01
 t3 :  2.66141319725174E-01   1.71943916132782E+00
 t6 : -1.05957839449340E+00   1.03531112257767E+00
 t4 :  2.42136902539447E-01   1.56435563278512E+00
 t5 : -8.79136932379912E-02  -5.67977370543055E-01
== err :  1.140E-15 = rco :  2.128E-02 = res :  4.441E-16 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  2.80836878017556E-01  -9.59755514673061E-01
 t2 : -9.99161738567148E-01   4.09367827689737E-02
 t3 : -4.72153176961024E-02   9.98884734979395E-01
 t6 : -9.37276397924231E-01   3.48587082225057E-01
 t4 :  9.35633636119240E-01  -3.52972660360955E-01
 t5 :  9.74098087752273E-01  -2.26125884049935E-01
== err :  2.120E-13 = rco :  2.187E-02 = res :  2.220E-16 =
Solution 4 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -1.24225818334613E+00  -8.07075027813304E-01
 t2 :  1.47332994921904E+00  -5.78910775622768E-01
 t3 :  8.79136932379910E-02   5.67977370543054E-01
 t6 : -4.82817017275396E-01   4.71759173981634E-01
 t4 :  9.66290808831228E-02   6.24284218493860E-01
 t5 : -2.66141319725174E-01  -1.71943916132781E+00
== err :  2.043E-13 = rco :  1.402E-02 = res :  2.637E-16 =
Solution 5 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  8.69193907041793E-01  -1.30453665759482E+00
 t2 : -2.10146778358548E-01  -5.53582709999063E-01
 t3 :  1.85739737768408E+00  -2.65498503011422E-01
 t6 : -4.40509781239126E-01  -4.61410384022367E-01
 t4 :  1.67183103323242E+00  -2.38973437749056E-01
 t5 : -5.27607584716191E-01   7.54168308853117E-02
== err :  6.392E-16 = rco :  1.136E-02 = res :  2.220E-16 =
Solution 6 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  3.53717078322846E-01  -5.30879118400565E-01
 t2 : -5.99365365166589E-01  -1.57888836421937E+00
 t3 :  5.27607584716190E-01  -7.54168308853108E-02
 t6 : -1.08247082572556E+00  -1.13383016827931E+00
 t4 :  5.86169848996842E-01  -8.37877878416832E-02
 t5 : -1.85739737768408E+00   2.65498503011425E-01
== err :  5.809E-16 = rco :  1.811E-02 = res :  6.661E-16 =
found 6 isolated solutions
```

The polynomials in the embedded system are printed by

```
for pol in embpols:
    print(pol)
```

with output in

```
+ 7.10358341606049E-01*t1 + 4.60000000000000E-01*t2 - 4.10000000000000E-01*t3 + (-3.
↪99034499614690E-01 + 9.16935912764493E-01*i)*zz1+(2.40761300555115E-01 + 1.
↪07248215701824E+00*i);
+ (-1.10000000000000E-01 + 4.90000000000000E-01*i)*t2 + 4.10000000000000E-01*t3 - 5.
↪02195181179589E-01*t4 + 4.10000000000000E-01*t5 + (-2.71603706128330E-01-9.
↪62409178477302E-01*i)*zz1;
+ 5.02195181179589E-01*t4 + (-9.80434782608696E-02 + 4.36739130434783E-01*i)*t5 - 7.
↪75518556663656E-01*t6 + (-9.98029449494905E-01 + 6.27472544490738E-02*i)*zz1 - 1.
↪20000000000000E+00;
+ (-9.58594885352021E-01-2.84773323499490E-01*i)*zz1+(2.40761300555115E-01-1.
↪07248215701824E+00*i) - 4.10000000000000E-01*t3^-1 + 4.60000000000000E-01*t2^-1 + 7.
↪10358341606049E-01*t1^-1;
+ (9.19472457329587E-01-3.93154422857342E-01*i)*zz1 + 4.10000000000000E-01*t5^-1 - 5.
↪02195181179589E-01*t4^-1 + 4.10000000000000E-01*t3^-1 + (-1.10000000000000E-01-4.
↪90000000000000E-01*i)*t2^-1;
+ (4.61876615285503E-01 + 8.86944187788841E-01*i)*zz1 - 1.20000000000000E+00 - 7.
↪75518556663656E-01*t6^-1 + (-9.80434782608696E-02-4.36739130434783E-01*i)*t5^-1 + 5.
↪02195181179589E-01*t4^-1;
+ (9.04264731472024E-01-4.26972241973442E-01*i)*t1 + (9.93763178327739E-01-1.
↪11511189572842E-01*i)*t2 + (8.26568833449879E-01 + 5.62835645254728E-01*i)*t3 + (8.
↪13748932516514E-01 + 5.81216547276687E-01*i)*t4 + (9.60611770393053E-01 + 2.
↪77893912459997E-01*i)*t5 + (-5.80987954131586E-02 + 9.98310838352234E-01*i)*t6 + (1.
↪86846899315945E-02 + 9.99825425943029E-01*i)*zz1+(-9.99671453748700E-01 + 2.
↪56317100475435E-02*i);
```

Observe that we have seven equation in seven variables, where the last variable is the slack variable `zz1`.

The code in

```
print('the isolated solutions :')
for (idx, sol) in enumerate(isosols):
    print('Solution', idx+1, ':')
    print(sol)
```

produces the following output:

```
the isolated solutions :
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -5.45421549881402E-01  -8.38161877518281E-01
 t2 : -5.49340801920855E-01  -8.35598398361888E-01
 t3 : -9.74098087752273E-01   2.26125884049935E-01
 t6 : -8.74935009483375E-01  -4.84240363022669E-01
 t4 :  1.78912046655936E-01  -9.83865071827120E-01
 t5 :  4.72153176961023E-02  -9.98884734979395E-01
== err :  3.858E-16 = rco :  3.659E-02 = res :  1.388E-16 =
```

(continues on next page)

(continued from previous page)

```
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -5.66058532229596E-01  -3.67759063106022E-01
 t2 :  5.87958784214732E-01  -2.31024744989669E-01
 t3 :  2.66141319725174E-01   1.71943916132782E+00
 t6 : -1.05957839449340E+00   1.03531112257767E+00
 t4 :  2.42136902539447E-01   1.56435563278512E+00
 t5 : -8.79136932379912E-02  -5.67977370543055E-01
== err :  1.140E-15 = rco :  2.128E-02 = res :  4.441E-16 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  2.80836878017556E-01  -9.59755514673061E-01
 t2 : -9.99161738567148E-01   4.09367827689737E-02
 t3 : -4.72153176961024E-02   9.98884734979395E-01
 t6 : -9.37276397924231E-01   3.48587082225057E-01
 t4 :  9.35633636119240E-01  -3.52972660360955E-01
 t5 :  9.74098087752273E-01  -2.26125884049935E-01
== err :  2.120E-13 = rco :  2.187E-02 = res :  2.220E-16 =
Solution 4 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 : -1.24225818334613E+00  -8.07075027813304E-01
 t2 :  1.47332994921904E+00  -5.78910775622768E-01
 t3 :  8.79136932379910E-02   5.67977370543054E-01
 t6 : -4.82817017275396E-01   4.71759173981634E-01
 t4 :  9.66290808831228E-02   6.24284218493860E-01
 t5 : -2.66141319725174E-01  -1.71943916132781E+00
== err :  2.043E-13 = rco :  1.402E-02 = res :  2.637E-16 =
Solution 5 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  8.69193907041793E-01  -1.30453665759482E+00
 t2 : -2.10146778358548E-01  -5.53582709999063E-01
 t3 :  1.85739737768408E+00  -2.65498503011422E-01
 t6 : -4.40509781239126E-01  -4.61410384022367E-01
 t4 :  1.67183103323242E+00  -2.38973437749056E-01
 t5 : -5.27607584716191E-01   7.54168308853117E-02
== err :  6.392E-16 = rco :  1.136E-02 = res :  2.220E-16 =
Solution 6 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 t1 :  3.53717078322846E-01  -5.30879118400565E-01
 t2 : -5.99365365166589E-01  -1.57888836421937E+00
 t3 :  5.27607584716190E-01  -7.54168308853108E-02
 t6 : -1.08247082572556E+00  -1.13383016827931E+00
```

(continues on next page)

(continued from previous page)

```
 t4 :  5.86169848996842E-01  -8.37877878416832E-02
 t5 : -1.85739737768408E+00   2.65498503011425E-01
== err :  5.809E-16 = rco :  1.811E-02 = res :  6.661E-16 =
```

```
fac = double_monodromy_breakup(embpols, sols0, 1, islaurent=True, verbose=True)
```

produces the following output:

```
... running monodromy loops in double precision ...
... initializing the grid for the linear trace ...
The diagnostics of the trace grid :
  largest error on the samples : 1.1914895647759973e-14
  smallest distance between the samples : 2.3530539196335214
... starting loop 1 ...
new permutation : [3, 2, 1]
number of factors : 3 -> 2
the decomposition :
  factor 1 : ([1, 3], 0.33179448122518296)
  factor 2 : ([2], 0.3317944812254451)
... starting loop 2 ...
new permutation : [2, 3, 1]
number of factors : 2 -> 1
the decomposition :
  factor 1 : ([1, 2, 3], 4.11226608321158e-13)
```

The grouping of the generic sets in one set shows that the solution curve is an irreducible cubic.

2.9 Tangent Lines via Witness Set Intersection

Let us compute the tangents lines to a circle via a witness set intersection. With matplotlib a plot is made of the tangent lines.

```
from math import cos, sin, pi
from random import uniform
import matplotlib.pyplot as plt
from phcpy.solver import solve
from phcpy.solutions import make_solution, coordinates, strsol2dict
from phcpy.sets import double_embed
from phcpy.diagonal import double_diagonal_solve
```

2.9.1 a witness set of the circle

Consider the following system of polynomial equations:

$$\begin{cases} (x-a)^2 + (y-b)^2 = r^2 \\ y = sx \end{cases}$$

which represents a circle centered at (a, b) with radius r, intersected by a line with slope s. The code below plots an example, a circle centered at $(3, 2)$, with radius 1, intersected with the line $y = x$.

Fig. 2.18 is made by the code below

```
fig1 = plt.figure()
axs = fig1.gca()
center = (3, 2)
radius = 1
circle = plt.Circle(center, radius, edgecolor='blue', facecolor='none')
axs.add_artist(circle)
plt.plot([0, 4], [0, 4], 'r')
plt.plot([2, 3], [2, 3], 'go')
plt.axis([0, 5, 0, 4])
plt.show()
```

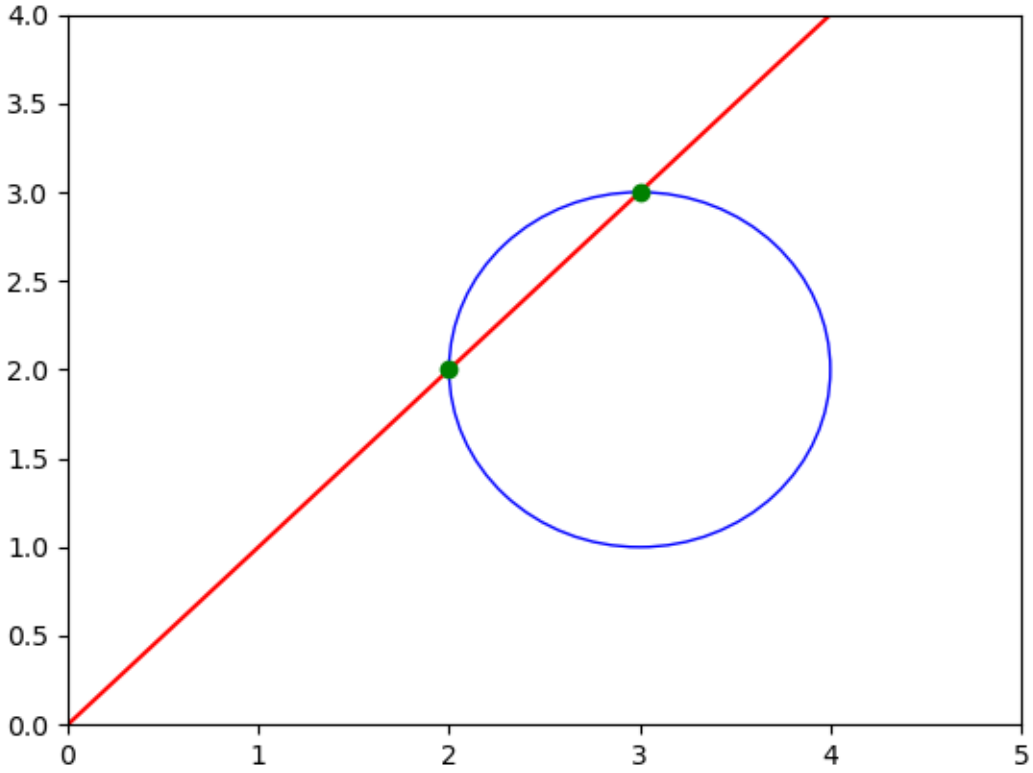


Fig. 2.18: A circle intersected with a line.

The system above has two equations in three variables (x, y, s) and thus defines a curve. What is the degree of this curve?

```
def polynomials(a, b, r):
    """
    Returns string representations of two polynomials:
    1) a circle with radius r centered at (a, b);
    2) a line through the origin with slope s.
    """
    crc = '(x - %.15e)^2 + (y - %.15e)^2 - %.15e;' % (a, b, r**2)
```

(continues on next page)

(continued from previous page)

```
    lin = 'y - s*x;'
    return [crc, lin]
```

```
def make_witness_set(pols, verbose=True):
    """
    We have two equations in three variables in pols
    and therefore we expect a one dimensional set.
    """
    embpols = double_embed(3, 1, pols)
    if verbose:
        print('the embedded system :')
        for pol in embpols:
            print(pol)
    embsols = solve(embpols)
    if verbose:
        print('the witness points :')
        for sol in embsols:
            print(sol)
    return (embpols, embsols)
```

```
def special_solutions(pols, slope):
    """
    Given in pols the polynomials for the line intersecting a circle
    for a general slope s, solves the problem for a special numerical
    value for s, given in the slope.
    The value of the slope is added as last coordinate to each solution.
    """
    special = []
    for pol in pols:
        rpl = pol.replace('s', '(' + str(slope) + ')')
        special.append(rpl)
    sols = solve(special)
    result = []
    for sol in sols:
        (vars, vals) = coordinates(sol)
        vars.append('s')
        vals.append(slope)
        extsol = make_solution(vars, vals)
        result.append(extsol)
    return result
```

Consider a circle with radius 1, centered at $(3, 2)$, intersected with the line $y = x$.

```
def circle_line_set():
    """
    Generates the system and its witness set.
    Returns the witness set for a fixed circle
    intersect with a one parameter family of lines
    as a tuple of polynomials and solutions.
    """
    syst = polynomials(3, 2, 1)
    for pol in syst:
```

(continues on next page)

(continued from previous page)

```
        print(pol)
    spsols = special_solutions(syst, 1)
    for (idx, sol) in enumerate(spsols):
        print('Solution', idx+1, ':')
        print(sol)
    (embsyst, embsols) = make_witness_set(syst, False)
    print('the polynomials in the witness set:')
    for pol in embsyst:
        print(pol)
    print('the solutions :')
    for (idx, sol) in enumerate(embsols):
        print('Solution', idx+1, ':')
        print(sol)
    print('degree of the set :', len(embsols))
    return (embsyst, embsols)
```

The output of

```
(witpols, witsols) = circle_line_set()
```

is

```
(x - 3.00000000000000e+00)^2 + (y - 2.00000000000000e+00)^2 - 1.00000000000000e+00;
y - s*x;
Solution 1 :
t : 0.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
 x : 3.00000000000000E+00  0.00000000000000E+00
 y : 3.00000000000000E+00  0.00000000000000E+00
 s : 1.00000000000000E+00  0.0
== err : 0.000E+00 = rco : 1.000E+00 = res : 0.000E+00 =
Solution 2 :
t : 0.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
 x : 2.00000000000000E+00  0.00000000000000E+00
 y : 2.00000000000000E+00  0.00000000000000E+00
 s : 1.00000000000000E+00  0.0
== err : 0.000E+00 = rco : 1.000E+00 = res : 0.000E+00 =
the polynomials in the witness set:
 + x^2 + y^2 - 6*x - 4*y + (-2.25888608394754E-01 + 9.74153138165392E-01*i)*zz1 + 12;
 - x*s + y + (4.03414981775719E-02 + 9.99185950424038E-01*i)*zz1;
zz1;
 + (-1.19285340138354E-01 + 9.92860014114818E-01*i)*x + (-8.44394833617596E-01-5.
↪35721350106483E-01*i)*y + (2.84736834956935E-01-9.58605724382401E-01*i)*s + (-8.
↪64325704104395E-01-5.02932477798403E-01*i)*zz1+(4.42123925817800E-01-8.96953975530214E-
↪01*i);
the solutions :
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
```

(continues on next page)

(continued from previous page)

```
the solution for t :
 x :  4.71992186119667E+00  -2.06479191154844E+00
 y :  4.21048963164569E+00   1.60655842789469E+00
 zz1 :  1.47124630003083E-32  -4.65633532178132E-32
 s :  6.23787940798390E-01   6.13262424188266E-01
== err :  3.253E-15 = rco :  4.216E-02 = res :  1.554E-15 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  2.22786304688919E+00  -3.25328777253638E-01
 y :  1.21728610339575E+00   3.20932555200177E-01
 zz1 :  8.77875924683185E-33  -8.43345264643039E-33
 s :  5.14387214185304E-01   2.19168552262572E-01
== err :  6.116E-16 = rco :  1.366E-01 = res :  2.220E-16 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -3.19419508001181E-01  -6.36087292918970E-01
 y :  1.33420600110974E+00   3.17131210618635E+00
 zz1 :  0.00000000000000E+00   0.00000000000000E+00
 s : -4.82279862042460E+00  -3.24310772611730E-01
== err :  3.542E-16 = rco :  7.767E-02 = res :  8.327E-16 =
Solution 4 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -1.02740100906214E+00  -1.33614560499347E+00
 y :  3.37463784588945E+00  -3.91462680435862E+00
 zz1 :  0.00000000000000E+00   0.00000000000000E+00
 s :  6.20734137916362E-01   3.00295170717090E+00
== err :  5.028E-16 = rco :  5.750E-02 = res :  1.110E-15 =\
degree of the set : 4
```

Thus, the degree of the algebraic curve defined by the system equals four.

2.9.2 intersecting with the Jacobian

For any random slope s, the line $y = sx$ will intersect the circle in exactly two (complex) points, as solutions of the polynomial system. A tangent line to the circle intersects the circle in one double solution, that is: a solution that has to be counted twice. We look for values of the slope for which the intersection points are singular. Singular points satisfy the original equations and the equations defined by all partial derivatives of the system, called the Jacobian.

First, a witness set for the Jacobian is constructed.

Let f_1 and f_2 denote the two polynomials in (x, y, s), then the Jacobian is

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} & \frac{\partial f_1}{\partial s} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} & \frac{\partial f_2}{\partial s} \end{bmatrix}$$

which for the polynomials in our problem becomes

$$J = \begin{bmatrix} 2(x-a) & 2(y-b) & 0 \\ -s & 1 & -x \end{bmatrix}$$

where the $-$ on the second row appears from the equation $-sx + y = 0$.

As we look for a singular solution, the rank of the Jacobian J must be one, not two. It suffices here to consider the first 2-by-2 minor and requiring that the first two columns are linearly dependent can be expressed by

$$\begin{cases} 2(x-a)\ L_1 + 2(y-b)\ L_2 = 0 \\ -s\ L_1 + L_2 = 0 \\ c_0 + c_1\ L_1 + c_2\ L_2 = 0 \end{cases}$$

where L_1 and L_2 are two new variables and c_0, c_1, and c_2 are random constants.

```
def random_complex():\n",
    """
    Returns a random complex number on the unit circle.
    """
    theta = uniform(0, 2*pi)\n",
    return complex(cos(theta), sin(theta))"
```

```
def random_hyperplane(vars):
    """
    Returns a linear equation in the variables
    in the list vars, with random complex coefficients.
    """
    cf0 = str(random_complex())
    tf0 = cf0.replace('j', '*i')
    result = tf0
    for var in vars:
        cff = str(random_complex())
        tcf = cff.replace('j', '*i')
        result = result + '+' + tcf + '*' + var
    return result + ';'
```

```
def jacobian(a, b):
    """
    Returns the equations which define the points
    where the Jacobian matrix is singular,
    as a random linear combination of the columns.
    Random complex coefficients are generated to
    scale the multiplier variables.
    """
    eq1 = '2*(x-%.15e)*L1 + 2*(y-%.15e)*L2;' % (a, b)
    eq2 = '-s*L1 + L2;'
    eq3 = random_hyperplane(['L1', 'L2'])
    return [eq1, eq2, eq3]
```

Then here is an example for the center $(3, 2)$.

```
jacpols = jacobian(3, 2)
for pol in jacpols:
    print(pol)
```

which produces the polynomials

```
2*(x-3.000000000000000e+00)*L1 + 2*(y-2.000000000000000e+00)*L2;
-s*L1 + L2;
(0.7467754449282183-0.6650762624333104*i)+(-0.05113246140388107-0.
↪9986918801065625*i)*L1+(0.8583591351915719-0.5130493105279226*i)*L2;
```

Let us now make a witness set, using the auxiliary function below.

```
def witset(pols, verbose=True):
    """
    We have three equations in pols in five variables:
    x, y, s, L1, and L2.
    Therefore we expect a two dimensional set.
    """
    embpols = double_embed(5, 2, pols)
    if verbose:
        print('the embedded system :')
        for pol in embpols:
            print(pol)
    embsols = solve(embpols)
    if verbose:
        print('the witness points :')
        for sol in embsols:
            print(sol)
    return (embpols, embsols)
```

As we have three equations in five variables (x, y, s, L_1, L_3), the solution set is two dimensional. The construction of this witness set will compute the degree of this two dimensional solution set

```
def singular_locus_set():
    """
    Generates a witness set for the singular locus
    of the algebraic set of a fixed circle intersected
    with a one parameter family of lines.
    """
    syst = jacobian(3, 2)
    for pol in syst:
        print(pol)
    (embsyst, embsols) = witset(syst, False)
    print('the polynomials in the witness set :')
    for pol in embsyst:
        print(pol)
    print('the solutions :')
    for sol in embsols:
        print(sol)
    print('degree of the singular locus set :', len(embsols))
    return (embsyst, embsols)
```

The output of

```
witset2 = singular_locus_set()
```

is

```
2*(x-3.000000000000000e+00)*L1 + 2*(y-2.000000000000000e+00)*L2;
-s*L1 + L2;
(-0.6742208883330486-0.7385297514219686*i)+(-0.9307410468108358-0.
↪36567896272751255*i)*L1+(0.4913219172246197+0.8709780557825346*i)*L2;
the polynomials in the witness set :
 + 2*x*L1 + 2*y*L2 - 6*L1 - 4*L2 + (4.93184439043922E-01 + 8.69924772083731E-01*i)*zz1 +␣
↪(4.77962297913147E-01 + 8.78380351427321E-01*i)*zz2;
 - L1*s + L2 + (-3.50715092194063E-03-9.99993849927294E-01*i)*zz1 + (6.78305496062243E-
↪01-7.34780003818663E-01*i)*zz2;
 + (-9.30741046810836E-01-3.65678962727513E-01*i)*L1 + (4.91321917224620E-01 + 8.
↪70978055782535E-01*i)*L2 + (-1.73576955750725E-01 + 9.84820308702207E-01*i)*zz1 + (-8.
↪69131200089510E-01 + 4.94581597950195E-01*i)*zz2+(-6.74220888333049E-01-7.
↪38529751421969E-01*i);
zz1;
zz2;
 + (-1.94485007680957E-01-2.95255113058755E-01*i)*x + (-9.05619807296798E-02-3.
↪41757995731361E-01*i)*L1 + (2.35555493200339E-01-2.63654337387317E-01*i)*y + (2.
↪27470946523190E-01 + 2.70660245488406E-01*i)*L2 + (3.53287256092446E-01 + 1.
↪37154906099080E-02*i)*s + (-2.50508068634215E-01-2.49490896726024E-01*i)*zz1 + (3.
↪37269024813124E-01 + 1.06064154649929E-01*i)*zz2+(-3.53553390593274E-01-1.
↪38777878078145E-17*i);
 + (-4.40566415427727E-01 + 1.40592718839002E-02*i)*x + (-2.45676058279196E-01-2.
↪88340078644854E-01*i)*L1 + (2.53550532539115E-01 + 6.86270383050135E-02*i)*y + (-2.
↪85183161836763E-01-2.03395928752962E-01*i)*L2 + (2.39045196476971E-01 + 1.
↪56472612484202E-01*i)*s + (-9.34949306113738E-02 + 4.61532488129392E-01*i)*zz1 + (-1.
↪98366575492475E-01 + 1.97233362690695E-01*i)*zz2+(2.06250450361319E-01-2.
↪15268649297659E-01*i);
the solutions :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.89091316713556E+00  -2.07966565089632E+00
 L1 : -1.55981268759518E-01   1.28195421958451E+00
 y :  9.29382853428914E-01   1.21952076881767E+00
 L2 :  1.66239264497059E+00   8.68576316936316E-01
 zz1 :  1.23844424223158E-33   2.14768913973412E-32
 zz2 : -3.68597450920629E-33  -1.56050619802567E-32
 s :  5.12174926048101E-01  -1.35908311946356E+00
== err :  1.287E-15 = rco :  3.154E-02 = res :  1.332E-15 =
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.30693118366194E+00  -1.84306554595479E-01
 L1 : -7.81141131311130E-01   2.88897684810126E-01
 y :  9.23707475868645E-01   1.58931255339032E+00
 L2 :  5.50791044105935E-01   4.92640130916949E-01
 zz1 : -5.01359801141478E-33  -1.18066233289204E-32
 zz2 :  0.00000000000000E+00   0.00000000000000E+00
 s : -4.15087884109074E-01  -7.84183593815662E-01
== err :  6.146E-16 = rco :  3.225E-02 = res :  8.882E-16 =
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
```

(continues on next page)

(continued from previous page)

```
 x :  2.56295575999062E-01   5.00768723718571E-01
 L1 : -1.54744896404770E-01  -3.07935271191708E-01
 y :  1.82176565498780E+00  -1.26116846711605E+00
 L2 :  6.60149130688637E-01  -3.65627094004125E-01
 zz1 : -8.24676473579955E-32  -3.81646685718875E-32
 zz2 :  0.00000000000000E+00   0.00000000000000E+00
 s :  8.78568617765025E-02   2.18794206021058E+00
== err :  8.086E-16 = rco :  2.783E-02 = res :  5.135E-16 =
degree of the singular locus set : 3
```

Indeed, the singular locus has degree three.

2.9.3 extending with slack variables

The first witness set of the circle intersected with a line is a cubic curve. In order to intersect this set with the singular locus, we have to add L_1 and L_2, increasing the dimension by two, using one as the values of the witness points for L_1 and L_2.

The function below adds to the solutions the values for the L_1 and L_2, and adds two additional slack variables zz2 and zz3.

```
def extend_solutions(sols):
    """
    To each solution in sols, adds L1 and L2 with values 1,
    and zz2 and zz3 with values zero.
    """
    result = []
    for sol in sols:
        (vars, vals) = coordinates(sol)
        vars = vars + ['L1', 'L2', 'zz2', 'zz3']
        vals = vals + [1, 1, 0, 0]
        extsol = make_solution(vars, vals)
        result.append(extsol)
    return result
```

```
def extend(pols, sols, verbose=True):
    """
    Extends the witness set with two free variables
    L1 and L2, addition two linear equations,
    and two slack variables zz2 and zz3.
    """
    vars = ['zz2', 'zz3']
    eq1 = 'zz2;'
    eq2 = 'zz3;'
    eq3 = 'L1 - 1;'
    eq4 = 'L2 - 1;'
    extpols = pols[:-1] + [eq1, eq2, eq3, eq4, pols[-1]]
    extsols = extend_solutions(sols)
    if verbose:
        print('the extended polynomials :')
        for pol in extpols:
```

(continues on next page)

(continued from previous page)

```
            print(pol)
        print('the extended solutions :')
        for sol in extsols:
            print(sol)
    return (extpols, extsols)
```

The output of

```
witset1 = extend(witpols, witsols)
```

is

```
the extended polynomials :
 + x^2 + y^2 - 6*x - 4*y + (-2.25888608394754E-01 + 9.74153138165392E-01*i)*zz1 + 12;
 - x*s + y + (4.03414981775719E-02 + 9.99185950424038E-01*i)*zz1;
zz1;
zz2;
zz3;
L1 - 1;
L2 - 1;
 + (-1.19285340138354E-01 + 9.92860014114818E-01*i)*x + (-8.44394833617596E-01-5.
↪35721350106483E-01*i)*y + (2.84736834956935E-01-9.58605724382401E-01*i)*s + (-8.
↪64325704104395E-01-5.02932477798403E-01*i)*zz1+(4.42123925817800E-01-8.96953975530214E-
↪01*i);
the extended solutions :
t : 0.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
 x : 4.71992186119667OE+00  -2.06479191154844O+00
 y : 4.21048963164569O+00  1.60655842789469O+00
 zz1 : 1.47124630003083O-32  -4.65633321781320-32
 s : 6.23787940798390O-01  6.13262424188266O-01
 L1 : 1.00000000000000E+00  0.0
 L2 : 1.00000000000000E+00  0.0
 zz2 : 0.00000000000000E+00  0.0
 zz3 : 0.00000000000000E+00  0.0
== err : 0.000E+00 = rco : 1.000E+00 = res : 0.000E+00 =
t : 0.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
 x : 2.22786304688919O+00  -3.25328777253638O-01
 y : 1.21728610339575O+00  3.20932555200177O-01
 zz1 : 8.778759246831849E-33  -8.433452646430390E-33
 s : 5.14387214185304O-01  2.19168522625720E-01
 L1 : 1.00000000000000E+00  0.0
 L2 : 1.00000000000000E+00  0.0
 zz2 : 0.00000000000000E+00  0.0
 zz3 : 0.00000000000000E+00  0.0
== err : 0.000E+00 = rco : 1.000E+00 = res : 0.000E+00 =
t : 0.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
```

(continues on next page)

(continued from previous page)

```
 x : -3.19419508001181OE-01  -6.360872929189700E-01
 y : 1.334206001109740E+00  3.171312106186350E+00
 zz1 : 0.000000000000000E+00  0.000000000000000E+00
 s : -4.822798620424600E+00  -3.243107726117300E-01
 L1 : 1.000000000000000E+00  0.0
 L2 : 1.000000000000000E+00  0.0
 zz2 : 0.000000000000000E+00  0.0
 zz3 : 0.000000000000000E+00  0.0
== err : 0.000E+00 = rco : 1.000E+00 = res : 0.000E+00 =
t : 0.000000000000000E+00 0.000000000000000E+00
m : 1
the solution for t :
 x : -1.027401009062140E+00  -1.336145604993470E+00
 y : 3.374637845889450E+00  -3.914626804358620E+00
 zz1 : 0.000000000000000E+00  0.000000000000000E+00
 s : 6.207341379163620E-01  3.002951707170900E+00
 L1 : 1.000000000000000E+00  0.0
 L2 : 1.000000000000000E+00  0.0
 zz2 : 0.000000000000000E+00  0.0
 zz3 : 0.000000000000000E+00  0.0
== err : 0.000E+00 = rco : 1.000E+00 = res : 0.000E+00 =
```

2.9.4 intersecting two witness sets

When intersecting two witness sets, the symbols must line up. The function below ensures that the order of symbols is $(x, y, s, L1, L2)$ by adding a zero polynomial to a given polynomial.

```
def insert_symbols(pol):
    """
    To the string pol, adds the sequence of symbols.
    """
    q = pol.lstrip()
    if q[0] == '+' or q[0] == '-':
        smb = 'x - x + y - y + s + L1 - L1 + L2 - L2 - s '
    else:
        smb = 'x - x + y - y + s + L1 - L1 + L2 - L2 - s + '
    return smb + pol
```

```
def intersect(dim, w1d, w2d, ws1, ws2, verbose=True):
    """
    Applies the diagonal homotopy to intersect two witness sets
    w1 and w2 of dimensions w1d and w2d in a space of dimension dim.
    """
    w1eqs, w1sols = ws1
    w2eqs, w2sols = ws2
    nw1eq0 = insert_symbols(w1eqs[0])
    nw2eq0 = insert_symbols(w2eqs[0])
    nw1eqs = [nw1eq0] + w1eqs[1:]
    nw2eqs = [nw2eq0] + w2eqs[1:]
    if verbose:
```

(continues on next page)

(continued from previous page)

```
        print('number of equations in first witness set :', len(w1eqs))
        print('number of equations in second witness set :', len(w2eqs))
    result = double_diagonal_solve(dim, w1d, nw1eqs, w1sols, w2d, nw2eqs, w2sols, \
        vrblvl=int(verbose))
    (eqs, sols) = result
    if verbose:
        print('the equations :')
        for pol in eqs:
            print(pol)
    return result
```

In a five dimensional space, we intersect a three dimensional set with a two dimensional one.

```
(eqs, sols) = intersect(5, 3, 2, witset1, witset2, verbose=False)
print('the solutions :')
for (idx, sol) in enumerate(sols):
    print('Solution', idx+1, ':')
    print(sol)
```

The output of the above code is

```
the solutions :
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  3.30216947925196E+00   1.91366021429925E-16
 y :  1.04674578112206E+00   6.06605980827076E-17
 s :  3.16987298107781E-01  -4.27079988595092E-32
 L1 : -3.06590066624447E-01  -2.66629188670126E-01
 L2 : -9.67199848241872E-01  -8.41135245045269E-01
== err :  2.341E-15 = rco :  1.884E-02 = res :  1.332E-15 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  2.23629205920958E+00   2.45132855242376E-17
 y :  2.64556191118564E+00   2.89995281402837E-17
 s :  1.18301270189222E+00   1.27877769258935E-33
 L1 :  5.35439409455237E-01  -1.48149502880241E+00
 L2 :  4.52606644543044E-01  -1.25230695024049E+00
== err :  6.349E-15 = rco :  2.003E-02 = res :  2.220E-16 =
```

Observe that we computed two real solutions, with the values of the intersection points and the slopes of the tangent lines. The two real solutions are regular solutions.

2.9.5 plotting the tangent lines

From the solution we extract the (real) coordinates of the intersection points and the slopes of the tangent lines, as codified in the following function.

```
def coordinates_and_slopes(sol):
    """
    Given a solution, return the 3-tuple with the x and y coordinates
    of the tangent point and the slope of the tangent line.
    The real parts of the coordinates are selected.
    """
    sdc = strsol2dict(sol)
    return (sdc['x'].real, sdc['y'].real, sdc['s'].real)
```

We apply the function to the first solution:

```
sol1 = sols[0]
coordinates_and_slopes(sol1)
```

with the output

```
(3.30216947925196, 1.04674578112206, 0.316987298107781)
```

and we do this also for the second solution

```
sol2 = sols[1]
coordinates_and_slopes(sol2)
```

which gives the output

```
(2.23629205920958, 2.64556191118564, 1.18301270189222)"
```

The code below plots the circle with radius 1, centered at (3, 2), along with its two lines tangent through (0, 0), with the plot shown in Fig. 2.19. The two solutions `sol1` and `sol2` define 3-tuples of x and y coordinates and a slope.

```
fig2 = plt.figure()
(xp1, yp1, s1) = coordinates_and_slopes(sol1)
(xp2, yp2, s2) = coordinates_and_slopes(sol2)
axs = fig2.gca()
center = (3, 2)
radius = 1
circle = plt.Circle(center, radius, edgecolor='blue', facecolor='none')
axs.add_artist(circle)
y1 = 5*s1 # first tangent line
y2 = 5*s2 # second tangent line
plt.plot([0, 5], [0, y1], 'r')
plt.plot([0, 5], [0, y2], 'r')
plt.plot([xp1, xp2], [yp1, yp2], 'go')
plt.axis([0, 5, 0, 4])
plt.show()
```

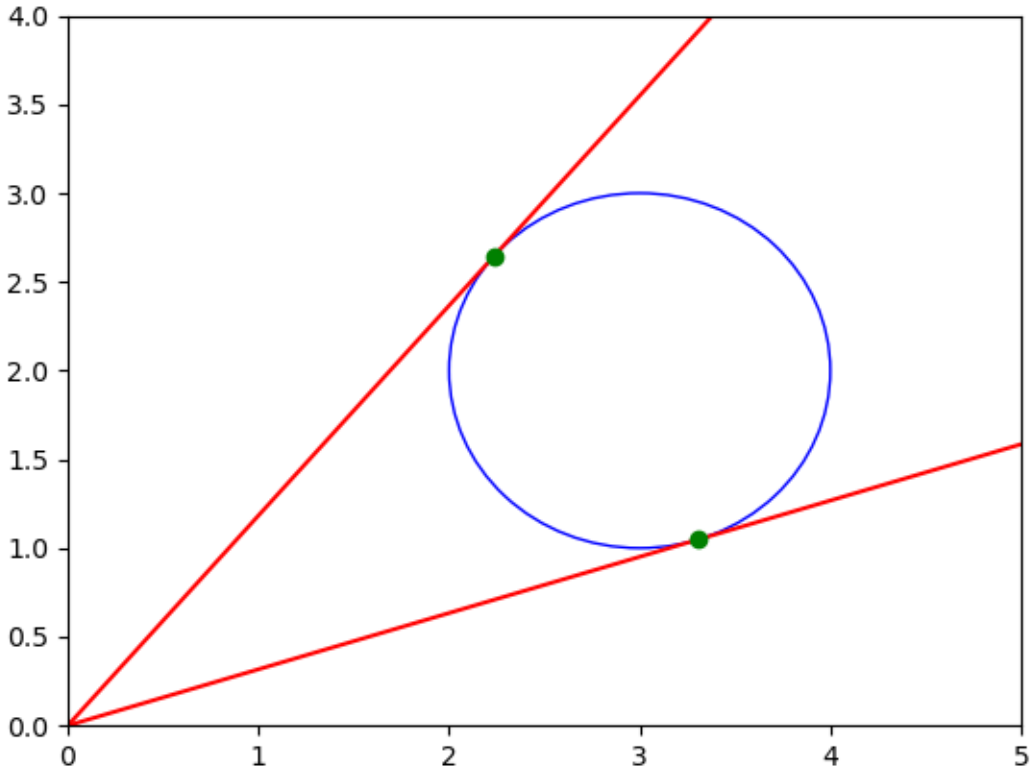


Fig. 2.19: Two lines tangent to a circle.

CHAPTER 3

User Manual

In solving a polynomial system, we first decide whether we care only about the isolated solutions, or whether we want to know all solutions, which includes not only the isolated solutions, but also all positive dimensional solutions.

3.1 The Blackbox Solver

A *blackbox* solver runs with default settings and values of the tolerances, executing a selection of algorithms which has shown to work for a large collection of problems. As to what the *solver* computes, two different types of solvers are available

1. to approximate all isolated solutions, typically of systems with as many equations as unknows; and
2. to compute a numerical irreducible decomposition of the entire solution set, which includes the isolated solution points, but also all positive dimensional sets (curves and surfaces), factored into irreducible components.

Because the output of the two types is so vastly different and because the complexity of a numerical irreducible decomposition is much higher than computing only the isolated solutions, the user must decide in advance which solver function to call.

3.1.1 approximating all isolated solutions

The input to the solver is a list of strings, with symbolic representations of the polynomials in the system. The `;` at the end signifies the `= 0` of the equations.

```
polynomials = ['x^3 + 2*x*y - x^2;', 'x + y - x^3;']
```

To call the blackbox solver, we import the `solve` function from the `solver` module.

```
from phcpy.solver import solve
solutions = solve(polynomials)
```

The output of `solve` is also a list of strings.

```
for (idx, sol) in enumerate(solutions):
    print('Solution', idx+1, ':')
    print(sol)
```

has as output

```
Solution 1 :
t :  0.00000000000000E+00   0.00000000000000E+00
m : 2
the solution for t :
 x :  0.00000000000000E+00   0.00000000000000E+00
 y :  0.00000000000000E+00   0.00000000000000E+00
== err :  7.124E-17 = rco :  0.000E+00 = res :  0.000E+00 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.00000000000000E+00   0.00000000000000E+00
 y :  0.00000000000000E+00   0.00000000000000E+00
== err :  2.100E-74 = rco :  5.348E-01 = res :  0.000E+00 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -1.50000000000000E+00   0.00000000000000E+00
 y : -1.87500000000000E+00   3.30053725596684E-238
== err :  6.747E-80 = rco :  1.220E-01 = res : 1.073E-237 =
```

How did this actually work? We can ask to see more output of the solver, giving a value to the verbose level parameter, the `vrblvl` as argument to the solver.

```
solutions = solve(polynomials, vrblvl=1)
```

shows the following

```
in solve, tasks : 0, mvfocus : 0, precision : d
the polynomials on input :
x^3 + 2*x*y - x^2;
x + y - x^3;
nbr : 4, roco :
total degree : 9
2-homogeneous Bezout number : 6
  with with partition : { x }{ y }
general linear-product Bezout number : 5
  based on the set structure :
     { x }{ x y }{ x }
     { x y }{ x }{ x }
mixed volume : 2
stable mixed volume : 4
```

The `roco` is an abbreviation of *root count*. A root count is an upper bound on the number of solutions. In the example, the two input polynomials are cubics. Therefore, the largest number of isolated solutions equals nine, the product of the degrees of the polynomials. However, not all monomials of degree three or less that could appear with nonzero coefficient are present. The numbers 6 and 5 are better bounds computed on the degrees. Mixed volumes are generically

sharp bounds for the number of isolated solutions with all coordinates different from zero. As in the example, there is one double root at `(0, 0)`, which is counted by the stable mixed volume.

If the coefficients of the polynomials are sufficiently independent from each other, then the number of isolated solutions counted with multiplicity will match one of the computed root counts.

3.1.2 options of the solve function

The blackbox solver runs with default values, which can be changed when calling the `solve` function.

In addition to the list of polynomials, there are six parameters with default values:

1. `tasks` equals the number of the threads and is by default set to zero. The solutions can approximated independently to each other using p threads could in the best case speed up the solver by a factor of p.
2. `mvfocus` is a flag to apply only mixed volumes as root counts, if set to one. By default, this flag is set to zero and the solver will compute all bounds based on the degrees which may be too time consuming for sparse polynomial systems.
3. `precision` is by default double (`d`). Other values are `dd` for double double and `qd` for quad double. While running in higher precision leads to more accurate results, the computational overhead can be significant. The overhead may be compensated (in part) by multithreading.
4. `checkin` is the option to run some basic checks on the input. By default `True`, setting it to `False` is okay if the input polynomials are automatically generated in the correct way.
5. `dictionary_output` is by default `False` and a list of strings is returned. If `True`, then the output is a list of dictionaries, often convenient for processing.
6. `vrblvl` is the verbose level parameter to make the blackbox solver less black when set to higher values.

Of course, then there is always the `help(solve)`:

```
help(solve)
```

which shows the documentation string of the function.

When changing the default values, consider the following.

1. The number of threads should never be set to a value higher than the number of available cores on the system. To find out the number of available cores, do the following:

   ```
   from phcpy.dimension import get_core_count
   get_core_count()
   ```

   So, use up to the value returned by `phcpy.dimension.get_core_count()` as the value to assign to the parameter `tasks`.
2. The focus on mixed volumes (in the option `mvfocus`) is automatically applied when the polynomials have negative exponents.
3. When computing in higher precision, keep in mind that also the coefficients of the polynomials then must also be evaluated in higher precision. Consider `1/3` in double precision:

   ```
   1/3
   ```

   which evaluates to

   ```
   0.3333333333333333
   ```

which is of course not equal to the rational number 1/3.

4. One of the checks done by default (checkin=True) is whether the number of polynomials in the list equals the number of unknowns. At this stage, if the syntax of the polynomial is incorrect, an error message will be printed as well.

5. If dictionary_output is wanted after a run, then it can be computed afterwards, the solve() should not be called again, but can be computed with the strsol2dict() of the solutions module. For example:

```
from phcpy.solutions import strsol2dict
strsol2dict(solutions[0])
```

shows the output

```
{'t': 0j, 'm': 2, 'err': 1.17e-16, 'rco': 0.0, 'res': 0.0, 'x': 0j, 'y': 0j}
```

6. Higher values of the verbose level vrblvl are mainly meant for debugging purposes as it should procedures are executed. As the solving of a polynomial system could take a long time, the user can see which procedures are currently running if the solver appears to be stuck.

3.2 Numerical Polynomials

If the input is wrong, then the output will be wrong as well.

3.2.1 symbols

The solver computes with complex numbers and the imaginary unit can be denoted by i or I. For example:

```
p = 'x - i;'
```

which represents the polynomial x - i. While i and I are symbols, they should not be used as the names of variables, because i and I are interpreted as numbers.

The polynomials can be written in factored form. For example:

```
p = '(y - 2)*(x + y - 1)^2;'
```

To enter this polynomial, first, the number of variables must be set.

```
from phcpy.dimension import set_double_dimension, get_double_dimension
set_double_dimension(2)
get_double_dimension()
```

which returns 2, confirming the number of variables.

```
from phcpy.polynomials import set_double_polynomial, get_double_polynomial
set_double_polynomial(1, 2, p)
get_double_polynomial(1)
```

shows

```
'y^3 + 2*y^2*x + y*x^2 - 4*y^2 - 6*y*x - 2*x^2 + 5*y + 4*x - 2;'
```

Observe that the variable y appears first in the polynomial.

```
from phcpy.polynomials import string_of_symbols
string_of_symbols()
```

confirms the list of symbols used as variable names:

```
['y', 'x']
```

If that is inconvenient, the simple trick is to add the null polynomial `x - x` in front of the string representation to ensure `x` comes first.

```
q = 'x - x + ' + p
q
```

defines the string `'x - x + (y - 2)*(x + y - 1)^2;'`

To continue, the polynomial that has been set must be cleared.

```
from phcpy.polynomials import clear_double_system
clear_double_system()
```

```
set_double_dimension(2)
set_double_polynomial(1, 2, q)
get_double_polynomial(1)
```

which shows

```
'x^2*y + 2*x*y^2 + y^3 - 2*x^2 - 6*x*y - 4*y^2 + 4*x + 5*y - 2;'
```

Polynomials added to the system will follow the same order of symbols.

```
r = 'y**2 + 4*x + y - 1;'
set_double_polynomial(2, 2, r)
get_double_polynomial(2)
```

confirms

```
'y^2 + 4*x + y - 1;'
```

Consider the entire system of polynomials:

```
from phcpy.polynomials import get_double_system
get_double_system()
```

returns the list

```
['x^2*y + 2*x*y^2 + y^3 - 2*x^2 - 6*x*y - 4*y^2 + 4*x + 5*y - 2;',
 'y^2 + 4*x + y - 1;']
```

3.2.2 numbers

The letters `e` and `E` appear in the scientific notation of floating-point numbers. Therefore, these letters `e` and `E` should not be used as the names of variables.

Even as coefficients can be entered as exact rational numbers, they are evaluated to floating-point numbers.

```
clear_double_system() # restart
```

```
p = 'x - 1/3;'
```

Entering `p` goes then as follows:

```
set_double_dimension(1)
set_double_polynomial(1, 1, p)
get_double_polynomial(1)
```

which then shows the decimal floating-point expansion of `1/3`:

```
' + x - 3.33333333333333E-01;'
```

Let us double the precision, first importing the corresponding double double functions:

```
from phcpy.dimension import set_double_double_dimension
from phcpy.polynomials import set_double_double_polynomial
```

The statements

```
from phcpy.polynomials import get_double_double_polynomial
set_double_double_dimension(1)
set_double_double_polynomial(1, 1, p)
get_double_double_polynomial(1)"
```

then show

```
' + x-3.33333333333333333333333333333324E-1;'
```

3.3 Numerical Solutions

Solutions of are numerical and returned as lists of PHCpack solution strings. The solutions module contains functions to parse a PHCpack solution string into a dictionary.

The solutions module exports operations

1. to parse strings in the PHCpack solution format into dictionaries;
2. to evaluate these dictionaries into polynomials substituting the values for the variables into the strings representing the polynomials.

Another useful operation is the `verify` function, to evaluate the polynomials at solutions.

3.3.1 attributes of numerical solutions

The information of a solution as a dictionary contains the following:

1. *t* : value of the continuation parameter
2. *m* : multiplicity of the solution
3. symbols for the variables are keys in the dictionary, the corresponding values are complex floating-point numbers
4. *err* : magnitude of the last correction term of Newton's method (forward error)

   rco : estimate for the inverse of the condition number of the Jacobian matrix at the solution

   res : magnitude of the residual (backward error)

The triplet (*err*, *rco*, *res*) measures the numerical quality of the solution. The residual *res* is normally interpreted as an estimate of the backward error: by how much should we change the original problem such that the approximate solution becomes an exact solution of the changed problem? The estimate *rco* gives a (sometimes too pessimistic) bound on the number of correct decimal places in the approximate solution. In particular: `abs(log(rco, 10))` bounds the number of lost decimal places in the approximate solution. For example, if *rco* equals `1.0E-8`, then the last 8 decimal places in the coordinates of the solution could be wrong.

The best numerically conditioned linear systems arise when the normals to the coefficient vectors of the linear equations are perpendicular to each other, as in the next session:

```
from phcpy.solver import solve
p = ['x + y - 1;', 'x - y - 1;']
s = solve(p)
print(s[0])
```

with the output of the print statement:

```
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.00000000000000E+00   0.00000000000000E+00
 y :  0.00000000000000E+00  -0.00000000000000E+00
== err :  2.220E-16 = rco :  5.000E-01 = res :  0.000E+00 =
```

The value of *rco* is `5.0E-1` which implies that the condition number is bounded by 2, as *rco* is an estimate for the inverse of the condition number. Roundoff errors are doubled at most.

At the opposite end of the best numerically conditioned linear systems are those where the the normals to the coefficient vectors of the linear equations are almost parallel to each other, as illustrated in the next example:

```
p = ['x + y - 1;', 'x + 0.999*y - 1;']
s = solve(p)
print(s[0])
```

and the output of the above code cell is

```
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.00000000000000E+00   0.00000000000000E+00
 y :  0.00000000000000E+00  -0.00000000000000E+00
== err :  2.220E-16 = rco :  2.501E-04 = res :  0.000E+00 =
```

The reported estimate for the inverse of the condition number *rco* is `2.5E-4`, which implies that the condition number is estimated at 4,000. Thus for this example, roundoff errors may magnify thousandfold. In the next example, the condition number becomes a 10-digit number:

```
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.00000000000000E+00   0.00000000000000E+00
 y :  0.00000000000000E+00  -0.00000000000000E+00
== err :  2.220E-16 = rco :  2.500E-10 = res :  0.000E+00 =
```

which is produced by the code in the cell below:

```
p = ['x + y - 1;', 'x + 0.999999999*y - 1;']
s = solve(p)
print(s[0])
```

Observe that the actual value of the solution remains `(1,0)`, which on the one hand indicates that the condition number is a pessimistic bound on the accuracy of the solution. But on the other hand, (1,0) may give the false security that the solution is right, because the problem on input is very close to a linear system which has infinitely many solutions (the line `x + y - 1 = 0`) and not the isolated point `(1,0)`.

For a solution of the example `noon` from the module `families`, we convert the PHCpack format solution string to a dictionary as follows:

```
from phcpy.families import noon
s = solve(noon(3))
print(s[0])
```

which shows

```
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x1 : -6.77804511269800E-01   5.27500584353303E-01
 x2 :  1.35560902253960E+00   2.32882178444166E-17
 x3 : -6.77804511269800E-01  -5.27500584353303E-01
== err :  1.601E-16 = rco :  2.303E-01 = res :  4.996E-16 =
```

To turn this string into a dictionary, do the following:

```
from phcpy.solutions import strsol2dict
d = strsol2dict(s[0])
d.keys()
```

which shows

```
dict_keys(['t', 'm', 'err', 'rco', 'res', 'x1', 'x2', 'x3'])
```

To select the value of the `x1` coordinate, which is

```
(-0.6778045112698+0.527500584353303j)
```

then just do `d['x1']`.

Observe that the values of the dictionary `d` are evaluated strings, parsed into Python objects.

By plain substitution of the values of the dictionary representation of the solution into the string representation of the polynomial system we can verify that the coordinates of the solution evaluate to numbers close to the numerical working precision:

```
from phcpy.solutions import evaluate
e = evaluate(noon(3), d)
for x in e: print(x)
```

shows

```
(-2.886579864025407e-15+6.661338147750939e-16j)
(-4.440892098500626e-16-1.475351643981535e-17j)
(-2.886579864025407e-15-6.661338147750939e-16j)
```

The `evaluate` is applied in the `verify` which computes the sum of all evaluated polynomials, in absolute value, summed over all solutions.

```
from phcpy.solutions import verify
err = verify(noon(3), s)
```

The number `err` can be abbreviated into `2.2e-13` which is close enough to zero.

3.3.2 filtering solution lists

The module exports function to filter regular solutions, solutions with zero coordinates or real solutions. The filtering of real solutions is illustrated in the session below. We first define one real solution and another with a coordinate that has a nonzero imaginary part.

```
from phcpy.solutions import make_solution
s0 = make_solution(['x', 'y'], [complex(1, 0), complex(0, 2)])
print(s0)
```

shows

```
t : 0.00000000000000E+00 0.00000000000000E+00
m : 1\n",
the solution for t :\n",
 x : 1.00000000000000E+00  0.00000000000000E+00
 y : 0.00000000000000E+00  2.00000000000000E+00
== err : 0.000E+00 = rco : 1.000E+00 = res : 0.000E+00 =
```

and the output

```
t : 0.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
 x : 2.00000000000000E+00  0.0
 y : 3.00000000000000E+00  0.0
== err : 0.000E+00 = rco : 1.000E+00 = res : 0.000E+00 =
```

is produced by the the statements

```
s1 = make_solution(['x', 'y'], [float(2), float(3)])
print(s1)
```

The filtering of real solutions (with respect to a given tolerance) is provided by the functions `is_real` (on one solution) and `filter_real` (on a list of solutions).

Observe the tolerance `1.0e-8` as the second argument in the application of the `is_real` function.

```
from phcpy.solutions import is_real, filter_real
is_real(s0, 1.0e-8)
```

with respect to the tolerance `1.0e-8`, `is_real` returns `False`, as `s0` is not a real solution. For `s1`, `is_real(s1, 1.0e-8)` returns `True`.

Putting `[s0, s1]` into a list, to illustrate the selection of the real solutions, with

```
realsols = filter_real([s0, s1], 1.0e-8, 'select')
for sol in realsols: print(sol)
```

shows then indeed

```
t : 0.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
 x : 2.00000000000000E+00  0.0
 y : 3.00000000000000E+00  0.0
== err : 0.000E+00 = rco : 1.000E+00 = res : 0.000E+00 =
```

The functions `filter_regular` and `filter_zero_coordinates` to filter the regular solutions and those solutions with zero coordinates respectively operate in a manner similar as `filter_real`.

Another application of `make_solution` is to turn the solution at the end of path (with value 1.0 for `t`) to a solution which can serve at the start of another path (with value 0.0 for `t`). This is illustrated in the session below. We start by solving a simple system.

```
p = ['x**2 - 3*y + 1;', 'x*y - 3;']
s = solve(p)
print(s[0])
```

which shows

```
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -9.60087560673590E-01   1.94043922153735E+00
 y : -6.14512082773443E-01  -1.24199437256077E+00
== err :  3.317E-16 = rco :  2.770E-01 = res :  4.441E-16 =
```

Then we import the functions `coordinates` and `make_solution` of the module `solutions`.

```
from phcpy.solutions import coordinates
(names, values) = coordinates(s[0])
names"
```

shows

```
['x', 'y']
```

and

```
values
```

shows

```
[(-0.96008756067359+1.94043922153735j), (-0.614512082773443-1.24199437256077j)]
```

With the `names` and the `value` we can reconstruct the solution string.

```
s0 = make_solution(names, values)
print(s0)
```

with output

```
t : 0.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
 x : -9.600875606735900E-01   1.940439221537350E+00
 y : -6.145120827734430E-01   -1.241994372560770E+00
== err : 0.000E+00 = rco : 1.000E+00 = res : 0.000E+00 =
```

Observe that also the diagnostics are set to the defaults.

3.4 Reproducible Runs

The correctness of a polynomial homotopy relies on the choice of random complex constants. Except for an algebraic set of bad choices of complex constants, the solution paths are free of singularities, except at the end of the paths, when the system that is solved has singular solutions.

For correctness, it is important that the random constants are generated *after* the user has provided the input.

3.4.1 solving the cyclic 5-roots system twice

The random choice of constants makes that the solutions are computed in a random order, as the experiment in this section shows.

```
from phcpy.families import cyclic
```

The cyclic 5-roots problem belongs to an academic family of benchmark problems. The code

```
c5 = cyclic(5)
for pol in c5:
    print(pol)
```

prints the polynomials:

```
x0 + x1 + x2 + x3 + x4;
x0*x1 + x1*x2 + x2*x3 + x3*x4 + x4*x0;
x0*x1*x2 + x1*x2*x3 + x2*x3*x4 + x3*x4*x0 + x4*x0*x1;
x0*x1*x2*x3 + x1*x2*x3*x4 + x2*x3*x4*x0 + x3*x4*x0*x1 + x4*x0*x1*x2;
x0*x1*x2*x3*x4 - 1;
```

Now let us solve the system twice. In each run we print the first solution.

```
from phcpy.solver import solve
```

The outcome of the first run

```
s1 = solve(c5)
print(s1[0])
```

is

```
t :  1.0000000000000E+00   0.0000000000000E+00
m : 1
the solution for t :
 x0 : -8.09016994374948E-01   5.87785252292473E-01
 x1 :  2.11803398874989E+00  -1.53884176858763E+00
 x2 :  3.09016994374947E-01  -2.24513988289793E-01
 x3 : -8.09016994374947E-01   5.87785252292473E-01
 x4 : -8.09016994374947E-01   5.87785252292473E-01
== err :  1.152E-15 = rco :  1.034E-01 = res :  1.665E-15 =
```

and the second run

```
s2 = solve(c5)
print(s2[0])
```

shows

```
t :  1.0000000000000E+00   0.0000000000000E+00
m : 1
the solution for t :
 x0 : -1.18033988749895E-01   3.63271264002680E-01
 x1 :  3.09016994374947E-01  -9.51056516295154E-01
 x2 :  3.09016994374947E-01  -9.51056516295154E-01
 x3 :  3.09016994374947E-01  -9.51056516295154E-01
 x4 : -8.09016994374947E-01   2.48989828488278E+00
== err :  6.682E-16 = rco :  4.527E-02 = res :  1.874E-15 =
```

The cyclic 5-roots problem has 70 different solutions and with high likelihood, the first solution will be different in each run.

3.4.2 fixing the seed

Fixing the seed of the random number generators makes the solver deterministic. Consider the following experiment.

```
from phcpy.dimension import set_seed
```

We set the seed to the value `2024` and print the first solution again after solving the system:

```
set_seed(2024)
s3 = solve(c5)
print(s3[0])
```

which shows

```
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x0 :  1.00000000000000E+00  -8.03364927637306E-17
 x1 :  3.09016994374947E-01   9.51056516295154E-01
 x2 : -8.09016994374947E-01   5.87785252292473E-01
 x3 : -8.09016994374947E-01  -5.87785252292473E-01
 x4 :  3.09016994374947E-01  -9.51056516295154E-01
== err :  6.315E-16 = rco :  2.393E-01 = res :  4.631E-16 =
```

Let us do it again:

```
set_seed(2024)
s4 = solve(c5)
print(s4[0])
```

which then shows

```
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x0 :  1.00000000000000E+00  -8.03364927637306E-17
 x1 :  3.09016994374947E-01   9.51056516295154E-01
 x2 : -8.09016994374947E-01   5.87785252292473E-01
 x3 : -8.09016994374947E-01  -5.87785252292473E-01
 x4 :  3.09016994374947E-01  -9.51056516295154E-01
== err :  6.315E-16 = rco :  2.393E-01 = res :  4.631E-16 =
```

And of course, the point is that we see twice the same first solution.

3.5 Equation and Variable Scaling

A polynomial system may have coefficients which vary widely in magnitude. This may lead to inaccurate solutions. Equation scaling multiplies all coefficients in the same equation by the same constant. Variable scaling replaces the original variables by new variables times some constant. Both equation and variable scaling can reduce the variability of the magnitudes of the coefficients of the system.

In this section, the following functions are applied:

```
from phcpy.solutions import verify
from phcpy.solver import solve
from phcpy.scaling import double_scale_system
from phcpy.scaling import double_scale_solutions
```

3.5.1 solving without scaling

Consider the polynomials below.

```
p = ['0.000001*x^2 + 0.000004*y^2 - 4;', '0.000002*y^2 - 0.001*x;']
```

Observe the variation in magnitude between the coefficients of the polynomials in p. First, the system is solved, without scaling the coefficients.

```
psols = solve(p)
for (idx, sol) in enumerate(psols):
    print('Solution', idx+1, ':')
    print(sol)
```

Then the solutions are

```
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.23606797749980E+03  -4.34288187660045E-10
 y : -7.86151377757428E+02   2.63023405572965E-10
== err :  2.598E-03 = rco :  4.053E-04 = res :  8.002E-10 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.23606797749981E+03   1.58919085844426E-11
 y :  7.86151377757436E+02   9.62482268770893E-12
== err :  5.733E-04 = rco :  4.053E-04 = res :  6.661E-11 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -3.23606797750004E+03  -1.73527763271358E-11
 y :  4.58877485485452E-12  -1.27201964951414E+03
== err :  1.310E-03 = rco :  2.761E-04 = res :  1.855E-10 =
Solution 4 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -3.23606797749979E+03   5.48229606785362E-14
 y :  1.44974035789780E-14   1.27201964951407E+03
== err :  6.525E-05 = rco :  2.761E-04 = res :  4.063E-14 =
```

Observe the magnitude of the values for the coordinates of the solutions and the estimates for the inverse condition numbers in rco. The forward errors err are rather large and the residual res not so close to the machine precision.

The function verify evaluates the polynomials in the system at the solutions and returns the sum of the absolute values.

```
verify(p, psols)
```

yields the number

```
4.3339840153217635e-12
```

With scaling this error can be reduced.

3.5.2 solving after scaling

Equation and variable scaling is applied in double precision.

```
(q, c) = double_scale_system(p)
```

Consider now the coefficients of the scaled system:

```
for pol in q:
    print(pol)
```

and here are the scaled polynomials:

```
9.99999999999994E-01*x^2 + 9.99999999999996E-01*y^2 - 1.00000000000000E+00;
  " + 9.99999999999996E-01*y^2 - 9.99999999999997E-01*x;
```

and scaling coefficients in c are

```
[3.30102999566398,
 0.0,
 2.999999999999999,
 0.0,
 -0.6020599913279623,
 0.0,
 -0.30102999566398136,
 0.0,
 0.04028876017114153,
 0.0]
```

We solve the scaled system:

```
qsols = solve(q)
for (idx, sol) in enumerate(qsols):
    print('Solution', idx+1, ':')
    print(sol)
```

which gives the solutions

```
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -1.61803398874990E+00   2.94545607917864E-90
 y :  1.47272803958932E-90   1.27201964951407E+00
== err :  1.475E-16 = rco :  2.268E-01 = res :  6.661E-16 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
```

(continues on next page)

(continued from previous page)

```
 x : -1.61803398874990E+00   2.94545607917864E-90
 y : -1.47272803958932E-90  -1.27201964951407E+00
== err :  1.475E-16 = rco :  2.268E-01 = res :  6.661E-16 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  6.18033988749897E-01  -2.92604772168262E-98
 y :  7.86151377757425E-01   0.00000000000000E+00
== err :  8.868E-17 = rco :  4.601E-01 = res :  1.110E-16 =
Solution 4 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  6.18033988749897E-01  -2.92604772168262E-98
 y : -7.86151377757425E-01   0.00000000000000E+00
== err :  8.868E-17 = rco :  4.601E-01 = res :  1.110E-16 =
```

All solutions of the scaled system are well conditioned with small forward and backward errors.

The scaling coefficients in `c` are used to bring the solutions of the scaled problem to the original coordinates.

```
ssols = double_scale_solutions(len(q), qsols, c)
```

The output of

```
for (idx, sol) in enumerate(ssols):
    print('Solution', idx+1, ':')
    print(sol)
```

is

```
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -3.23606797749979E+03   5.89091215835726E-87
 y :  1.47272803958932E-87   1.27201964951407E+03
== err :  1.475E-16 = rco :  2.268E-01 = res :  6.661E-16 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -3.23606797749979E+03   5.89091215835726E-87
 y : -1.47272803958932E-87  -1.27201964951407E+03
== err :  1.475E-16 = rco :  2.268E-01 = res :  6.661E-16 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.23606797749979E+03  -5.85209544336522E-95
 y :  7.86151377757424E+02   0.00000000000000E+00
== err :  8.868E-17 = rco :  4.601E-01 = res :  1.110E-16 =
```

(continues on next page)

(continued from previous page)

```
Solution 4 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.23606797749979E+03  -5.85209544336522E-95
 y : -7.86151377757424E+02   0.00000000000000E+00
== err :  8.868E-17 = rco :  4.601E-01 = res :  1.110E-16 =
```

Let us look at the sum of the backward errors. Executing

```
verify(p, ssols)
```

yields the error

```
4.4853010194856324e-14
```

Observe that the error is about one hundred times smaller than without scaling."

3.6 Parallel Runs

Almost all computers are parallel and have multiple cores available. In a polynomial homotopy, all solution paths can be computed independently from each other.

```
from phcpy.dimension import get_core_count
nbcores = get_core_count()
nbcores
```

In the experiment we use the cyclic 7-roots problem.

```
from phcpy.families import cyclic
c7 = cyclic(7)
for pol in c7:
    print(pol)
```

which shows the polynomials

```
x0 + x1 + x2 + x3 + x4 + x5 + x6;
x0*x1 + x1*x2 + x2*x3 + x3*x4 + x4*x5 + x5*x6 + x6*x0;
x0*x1*x2 + x1*x2*x3 + x2*x3*x4 + x3*x4*x5 + x4*x5*x6 + x5*x6*x0 + x6*x0*x1;
x0*x1*x2*x3 + x1*x2*x3*x4 + x2*x3*x4*x5 + x3*x4*x5*x6 + x4*x5*x6*x0 + x5*x6*x0*x1 +␣
↪x6*x0*x1*x2;
x0*x1*x2*x3*x4 + x1*x2*x3*x4*x5 + x2*x3*x4*x5*x6 + x3*x4*x5*x6*x0 + x4*x5*x6*x0*x1 +␣
↪x5*x6*x0*x1*x2 + x6*x0*x1*x2*x3;
x0*x1*x2*x3*x4*x5 + x1*x2*x3*x4*x5*x6 + x2*x3*x4*x5*x6*x0 + x3*x4*x5*x6*x0*x1 +␣
↪x4*x5*x6*x0*x1*x2 + x5*x6*x0*x1*x2*x3 + x6*x0*x1*x2*x3*x4;
x0*x1*x2*x3*x4*x5*x6 - 1;
```

The mixed volume provides a generically sharp upper bound on the number of isolated solutions.

```
from phcpy.volumes import mixed_volume
mixed_volume(c7)
```

For this problem, there are as many solutions as the mixed volume. So, there are 924 paths to track.

```
from phcpy.solver import solve
```

3.6.1 speeding up the solver on multiple cores

To measure the speedup, the elapsed time between the start and the end of the run has to be computed. The most honest time measurement is the *wall clock time* which as suggested uses the time on the wall clock. The timers provided by Python do not measure the CPU time of compiled code that is executed by the solver.

```
from datetime import datetime
```

Then the code cell below does a run on one core and prints the elapsed wall clock time.

```
timestart = datetime.now()
s = solve(c7)
timestop = datetime.now()
elapsed_onecore = timestop - timestart
print('elapsed wall clock time on 1 core :', elapsed_onecore)
```

We check on `len(s)` whether we have as many solutions as the mixed volume.

Now we solve again, using all available cores, computed earlier in `nbcores`.

```
timestart = datetime.now()
s = solve(c7, tasks=nbcores)
timestop = datetime.now()
elapsed_manycores = timestop - timestart
print('elapsed wall clock time on', nbcores, 'cores:', elapsed_manycores)
```

We observe the reduction in the elapsed wall clock time and compute the speedup as follows.

```
speedup = elapsed_onecore/elapsed_manycores
speedup
```

3.6.2 quality up

Can multithreading compensate for the overhead of double double arithmetic? If we can afford the time for a sequential run, by how much can we increase the precision in a multithreaded run in the same time or less?

The code cell below computes all solutions in double double precision, on all avaible cores.

```
timestart = datetime.now()
s = solve(c7, tasks=nbcores, precision='dd')
timestop = datetime.now()
elapsed = timestop - timestart
print('elasped wall clock time on', nbcores, 'cores :', elapsed)
```

Again, we check whether we have as many solutions as the mixed volume.

If `elapsed < elapsed_onecore` evaluates to `True`, then we have achieved quality up in the multicore run. With the multicore run, we compensate for the cost overhead of double double arithmetic, if the elapsed wall clock time on many cores in double double precision is less than the run on one core in double precision."

3.7 Root Counts and Start Systems

A formal root count can be viewed as a count on the number of solutions of a system with a particular structure. The structure can be determined by the degrees or the Newton polytopes.

Each of the four root counts below is illustrated by a proper example.

3.7.1 total degree

The *total degree* is the product of the degrees of all polynomials in the system.

```
from phcpy.starters import total_degree
from phcpy.starters import total_degree_start_system
```

One family of polynomial systems for which the total degree equals the number of solutions is the Katsura benchmark problem.

```
from phcpy.families import katsura
k3 = katsura(3)
for pol in k3:
    print(pol)
```

Our example consists of one linear and three quadrics.

```
u3 + u2 + u1 + u0 + u1 + u2 + u3 - 1;
u3*u3 + u2*u2 + u1*u1 + u0*u0 + u1*u1 + u2*u2 + u3*u3 - u0;
u3*u3 + u2*u3 + u1*u2 + u0*u1 + u1*u0 + u2*u1 + u3*u2 - u1;
u3*u3 + u2 + u1*u3 + u0*u2 + u1*u1 + u2*u0 + u3*u1 - u2;
```

The total degree is 8, computed with

```
degk3 = total_degree(k3)
```

and the corresponding start system has also 8 solutions, as can be seen from the output of the code cell below.

```
q, qsols = total_degree_start_system(k3)
len(qsols)
```

The polynomials in the start system q are shown as

```
u3^1 - 1;
u2^2 - 1;
u1^2 - 1;
u0^2 - 1;
```

as output of

```
for pol in q:
    print(pol)
```

3.7.2 multihomogeneous Bezout numbers

Most polynomial systems arising in application have far fewer solutions than the total degree.

```
from phcpy.starters import m_homogeneous_bezout_number
from phcpy.starters import m_homogeneous_start_system"
```

We illustrate m-homogenous Bezout numbers with an application from game theory.

```
from phcpy.families import generic_nash_system
game4two = generic_nash_system(4)
for pol in game4two:
    print(pol)"
```

The (omitted) output of the code cell above shows four cubics. The output of

```
mbn = m_homogeneous_bezout_number(game4two)
mbn
```

is the tuple

```
(9, '{ p2 }{ p3 }{ p4 }{ p1 }')
```

The tuple on return contains first the root count, and then the partition of the variables. Observe the difference with the total degree, which is 81.

To construct a start system with 9 solutions, respecting the structure of the `game4two` system, we do:

```
q, qsols = m_homogeneous_start_system(game4two, partition=mbn[1])
len(qsols)
```

and we see 9 as the output of `len(qsols)`.

3.7.3 linear-product start systems

The multihomogeneous Bezout numbers are computed based on a partition of the unknowns. This partition is the same for all polynomials in the system. A sharper bound can be obtained if this restriction is relaxed.

```
from phcpy.starters import linear_product_root_count
from phcpy.starters import random_linear_product_system
```

The example we use to illustrate this root count consists of cubics.

```
from phcpy.families import noon
n3 = noon(3)
for pol in n3:
    print(pol)
```

In dimension three, the system is then

```
x1*(x2^2 + x3^2) - 1.1*x1 + 1;
x2*(x1^2 + x3^2) - 1.1*x2 + 1;
x3*(x1^2 + x2^2) - 1.1*x3 + 1;
```

The linear-product root count for this system is computed by the instructions in the cell below:

```
lprc = linear_product_root_count(n3)
lprc
```

which gives as output the tuple

```
(21,
'{ x1 }{ x2 x3 }{ x2 x3 };{ x1 x3 }{ x1 x3 }{ x2 };{ x1 x2 }{ x1 x2 }{ x3 };')
```

The tuple on return contains first the upper bound on the number of solutions, followed by the set structure used to compute this bound. Every set corresponds to a linear equation in the linear-product start system.

The start system is constructed and solved by the following:

```
q, qsols = random_linear_product_system(n3, lprc[1])
len(qsols)
```

and `len(qsols)` returns `21`.

3.7.4 mixed volumes

For sparse polynomial systems, that are systems with relatively few monomials appearing with nonzero coefficients, the mixed volume of the Newton polytopes provides a much sharper bound than any of the degree bounds.

```
from phcpy.volumes import mixed_volume
from phcpy.volumes import make_random_coefficient_system
```

The cyclic n-roots problem illustrates the need to apply mixed volumes very well.

```
from phcpy.families import cyclic
c5 = cyclic(5)
for pol in c5:
    print(pol)
```

which shows

```
x0 + x1 + x2 + x3 + x4;
x0*x1 + x1*x2 + x2*x3 + x3*x4 + x4*x0;
x0*x1*x2 + x1*x2*x3 + x2*x3*x4 + x3*x4*x0 + x4*x0*x1;
x0*x1*x2*x3 + x1*x2*x3*x4 + x2*x3*x4*x0 + x3*x4*x0*x1 + x4*x0*x1*x2;
x0*x1*x2*x3*x4 - 1;
```

The mixed volume equals `70` for this system, and is computed via

```
mv = mixed_volume(c5)
```

A *random coefficient system* has the same monomial structure as the input system, but random coefficients. The start system is made and solved with the code below

```
vol, q, qsols = make_random_coefficient_system(c5)
len(qsols)
```

and we see `70` as the output of `len(qsols)`.

The mixed volume does not count the solutions with zero coordinates. To count all solutions, also those with zero coordinates, the stable mixed volume should be computed.

```
from phcpy.volumes import stable_mixed_volume
```

Consider the following example.

```
pols = ['x^3 + x*y^2 - x^2;', 'x + y - y^3;']
```

The command

```
stable_mixed_volume(pols)
```

returns a tuple of two integers. The first number in the tuple is the mixed volume, the second number is the stable mixed volume, which takes into account the solutions with zero coordinates.

3.8 Increment-and-Fix Aposteriori Step Control

An *artificial-parameter homotopy* to solve

$$\begin{cases} x^2 + 4y^2 - 4 = 0 \\ 2y^2 - x = 0 \end{cases}$$

with the total degree start system

$$\begin{cases} x^2 - 1 = 0 \\ y^2 - 1 = 0 \end{cases}$$

uses a random complex constant γ in

$$\gamma(1-t)\begin{pmatrix} x^2 + 4y^2 - 4 \\ 2y^2 - x \end{pmatrix} + t\begin{pmatrix} x^2 - 1 \\ y^2 - 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

where t is an artificial parameter to define the deformation of the start system into the target system.

During the continuation, a predictor-corrector method is applied. The predictor advances the value of t, predicts the corresponding solution for the new value of t, and then fixes the value of t in the corrector stage. Because t is fixed during the corrector stage, this type of continuation is called *increment-and-fix continuation*.

An *aposteriori step size control algorithm* uses the performance of the corrector to determine the step size for the continuation parameter t.

Let us first define the target and start system for the running example.

```
target = ['x^2 + 4*y^2 - 4;', '2*y^2 - x;']
```

A start system based on the total degree is constructed.

```
from phcpy.starters import total_degree_start_system
start, startsols = total_degree_start_system(target)
```

The `len(startsols)` returns 4 and thus we have 4 paths.

For deterministic runs, we set the seed of the random number generators:

```
from phcpy.dimension import set_seed
set_seed(2024)
```

3.8.1 let the path trackers run

To run the path trackers in double precision:

```
from phcpy.trackers import double_track
```

and then call double_track as follows:

```
gamma, sols = double_track(target, start, startsols)
```

By default, double_track will generate a random $gamma$ and return the generated value. This value can then be used in a second run.

To print the solutions, execute the following code:

```
for (idx, sol) in enumerate(sols):
    print('Solution', idx+1, ':')
    print(sol)
```

and what is then printed is

```
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.23606797749979E+00   1.63132611699963E-55
 y :  7.86151377757423E-01  -1.62115537155392E-56
== err :  1.383E-16 = rco :  1.998E-01 = res :  2.220E-16 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.23606797749979E+00   1.63132611699963E-55
 y : -7.86151377757423E-01   1.62115537155392E-56
== err :  1.383E-16 = rco :  1.998E-01 = res :  2.220E-16 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -3.23606797749979E+00   1.06910588403688E-50
 y : -5.34552942018439E-51  -1.27201964951407E+00
== err :  7.222E-35 = rco :  1.079E-01 = res :  3.130E-50 =
Solution 4 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -3.23606797749979E+00   1.06910588403688E-50
 y :  5.34552942018439E-51   1.27201964951407E+00
"== err :  7.222E-35 = rco :  1.079E-01 = res :  3.130E-50 =
```

Of the four solutions, observe that two are complex conjugated. The start solutions are all real and if the γ would have been real as well (with a zero imaginary part), then the transition from the real start solutions to the solutions with nonzero imaginary parts would not have been possible.

Suppose we would want to recompute the first path in quad double precision.

```
from phcpy.trackers import quad_double_track
```

Even if we track only one path, the start solution must be given in a list of one element. Observe that we use the same value of gamma as before.

```
gamma, qdsol = quad_double_track(target, start, [startsols[0]], gamma=gamma)
print(qdsol[0])
```

and what is printed is

```
t : 1.00000000000000000000000000000000000000000000000000000000000000E+00      0.
↪00000000000000000000000000000000000000000000000000000000000000E+00
m : 1
the solution for t :
 x : 1.23606797749978969640917366871164299374027347444922679642033215 08E+00      3.
↪57254075854783981681880689384762092572682223811067256053578692 23E-29
 y : 7.86151377757423286069558585822729878807396336191494012382328 06827E-01      1.
↪97752783446602867322403685627382546091435800793784035948560659 32E-29
== err :  1.850E-14 = rco :  1.998E-01 = res :  3.884E-28 =
```

Observe that the values for the forward and backward error, the `err` and `res` respectively, are still rather large for quad double precision. For this example, we could as well run a couple of extra steps of Newton's method, but suppose that we want to track the complete path with much smaller tolerances.

3.8.2 tuning tolerances of the path trackers

Let us redo the last run, but now with much smaller tolerances on the corrector. The output of

```
from phcpy.trackers import write_parameters
write_parameters()
```

is

```
GLOBAL MONITOR :
  1. the condition of the homotopy           : 0
  2. number of paths tracked simultaneously  : 1
  3. maximum number of steps along a path    : 500
  4. distance from target to start end game  : 1.000e-01
  5. order of extrapolator in end game       : 0
  6. maximum number of re-runs               : 1
STEP CONTROL (PREDICTOR) :                    along path : end game
  7: 8. type ( x:Cub,t:Rea ):( x:Cub,t:Rea ) : 8         : 8
  9:10. minimum step size                    : 1.000e-06 : 1.000e-08
 11:12. maximum step size                    : 1.000e-01 : 1.000e-02
 13:14. reduction factor for step size       : 7.000e-01 : 5.000e-01
 15:16. expansion factor for step size       : 1.250e+00 : 1.100e+00
 17:18. expansion threshold                  : 1         : 3
PATH CLOSENESS (CORRECTOR) :                  along path : end game
 19:20. maximum number of iterations         : 3         : 3
 21:22. relative precision for residuals     : 1.000e-09 : 1.000e-11
 23:24. absolute precision for residuals     : 1.000e-09 : 1.000e-11
 25:26. relative precision for corrections   : 1.000e-09 : 1.000e-11
 27:28. absolute precision for corrections   : 1.000e-09 : 1.000e-11
```

(continues on next page)

(continued from previous page)

```
SOLUTION TOLERANCES :                          along path : end game,
 29:30. inverse condition of Jacobian        : 1.000e-04 : 1.000e-12
 31:32. clustering of solutions              : 1.000e-04 : 1.000e-12
 33:34. solution at infinity                 : 1.000e+08 : 1.000e+12
```

Let us tune of the parameters of the corrector. To set the tolerance for the relative precision for the residuals along the path to `1.0e-32`, the parameter at position `21` has to be set, as follows:

```
set_parameter_value(21, 1.0e-32)
```

and then running `write_parameters()` again will show

```
21:22. relative precision for residuals     : 1.000e-32 : 1.000e-11
```

as the line that has changed.

For this problem, the difference between absolute and relative does not matter, and neither does the difference between the residuals and corrections, as the paths are well conditioned. Along the path, we set the tolerance to `1.0e-32` and at the end to `1.0e-48`.

```
for idx in [23, 25, 27]:
    set_parameter_value(idx, 1.0e-32)
for idx in [22, 24, 26, 28]:
    set_parameter_value(idx, 1.0e-48)
```

Now we rerun the first path once more.

```
gamma, qdsol = quad_double_track(target, start, [startsols[0]], gamma=gamma)
print(qdsol[0])
```

and what is printed is

```
t : 1.00000000000000000000000000000000000000000000000000000000000E+00      0.
↪00000000000000000000000000000000000000000000000000000000000E+00
m : 1
the solution for t :
 x : 1.23606797749978969640917366873127623544061835961152572427089724540E+00      -5.
↪24972000008925235539408771989540466194549407263893577240824899420E-135
 y : 7.86151377757423286069558585842958929523122057837723237664901970120E-01      -5.
↪86376277993301867739637842830911009945665896835007407540444559870E-133
== err :  3.218E-66 = rco :  1.998E-01 = res :  2.018E-65 =
```

Observe that the values of `err` and `res` (forward and backward error respectively) are much smaller than before, very close to the quad double precision.

For the experiments in the next section, the values of the continuation parameters must be reset to their defaults.

```
from phcpy.trackers import autotune_parameters
autotune_parameters(0, 14)
```

The `0` stands for the default and `14` to take into account the 14 decimal places of precision.

3.8.3 a step-by-step path tracker

When we run a path tracker, or let a path tracker run, then the path tracker has the control of the order of execution. In a step-by-step path tracker, we can ask the path tracker for the next point of the path, which is useful to plot the points along a path.

```
from phcpy.trackers import initialize_double_tracker
from phcpy.trackers import initialize_double_solution
from phcpy.trackers import next_double_solution
```

The initialization of the tracker is separate from the initialization of a solution, as the same homotopy is used to track all paths.

```
initialize_double_tracker(target, start)
```

The first parameter in `initialize_double_solution` is the number of variables, which equals the number of polynomials in the target system.

```
initialize_double_solution(len(target), startsols[0])
```

The first step

::
    nextsol = next_double_solution() print(nextsol)

shows

```
t :  1.00000000000000E-01   0.00000000000000E+00
m : 1
the solution for t :
 x :  9.96326698649568E-01   4.70406409720798E-03
 y :  9.96408257454631E-01   4.95315220446915E-03
== err :  2.375E-05 = rco :  1.000E+00 = res :  3.619E-10 =
```

and then the second step

```
nextsol = next_double_solution()
print(nextsol)
```

gives

```
t :  2.00000000000000E-01   0.00000000000000E+00
m : 1
the solution for t :
 x :  9.79864035891029E-01   1.70985015865591E-02
 y :  9.81181263858417E-01   2.32157127720825E-02
== err :  1.679E-08 = rco :  1.000E+00 = res :  2.760E-16 =
```

In a loop, we want to stop as soon as the value of `t` passes `1.0`. To get the value of `t` out of a solution string, we convert the string into a dictionary, as done below:

```
from phcpy.solutions import strsol2dict
dictsol = strsol2dict(nextsol)
dictsol['t']
```

shows

```
(0.2+0j)
```

In the code cell below, the loop continues calling `next_double_solution` until the value of the continuation parameter is less than 1.0. The real part and imaginary part of the gamma constant are set to the same value of `gamma` as in the first run.

```
initialize_double_tracker(target, start, fixedgamma=False, \
                          regamma=gamma.real, imgamma=gamma.imag)
initialize_double_solution(len(target), startsols[0])
tval = 0.0
path = [startsols[0]]
while tval < 1.0:
    nextsol = next_double_solution()
    dictsol = strsol2dict(nextsol)
    tval = dictsol['t'].real
    path.append(nextsol)
```

All values of the x-coordinates of all points on the path:

```
(1+0j)
(0.996326698649568+0.0047040640972079 8j)
(0.979864035891029+0.0170985015865591j)
(0.943788099865787+0.00964655321202273j)
(0.950990471736517-0.0674242744055464j)
(1.06214893672862-0.108107883550403j)
(1.15606754413692-0.0765694315601522j)
(1.20399680236398-0.0383121382228797j)
(1.2254591757779-0.0141573296657659j)
(1.23400325696781-0.00289013787311083j)
(1.23606797749301-4.13424477147656e-11j)
```

are obtained with

```
for sol in path:
    print(strsol2dict(sol)['x'])
```

To put the real parts of the x-coordinates in a list:

```
xre = [strsol2dict(sol)['x'].real for sol in path]
```

and likewise, the imaginary parts of the x-coordinates and the two parts of the y-coordinates are set by the code below:

```
xim = [strsol2dict(sol)['x'].imag for sol in path]
yre = [strsol2dict(sol)['y'].real for sol in path]
yim = [strsol2dict(sol)['y'].imag for sol in path]
```

Let us plot the coordinates of this first solution path.

```
import matplotlib.pyplot as plt
```

The coordinates of the solution path are then plotted in Fig. 3.1 as follows.

```
fig, axs = plt.subplots(1, 2, constrained_layout=True)
fig.suptitle('the coordinates of one solution path')
```

(continues on next page)

(continued from previous page)

```
axs[0].set_title('x coordinates')
axs[0].set_xlabel('real part')
axs[0].set_ylabel('imaginary part')
axs[0].set_xlim(min(xre)-0.1, max(xre)+0.1)
axs[0].set_ylim(min(xim)-0.025, max(xim)+0.025)
dots, = axs[0].plot(xre,xim,'r-')
dots, = axs[0].plot(xre,xim,'ro')
axs[1].set_title('y coordinates')
axs[1].set_xlabel('real part')
axs[1].set_ylabel('imaginary part')
axs[1].set_xlim(min(yre)-0.1, max(yre)+0.1)
axs[1].set_ylim(min(yim)-0.025, max(yim)+0.025)
dots, = axs[1].plot(yre,yim,'r-')
dots, = axs[1].plot(yre,yim,'ro')
plt.savefig('incfixaposteriorifig1')
plt.show()
```

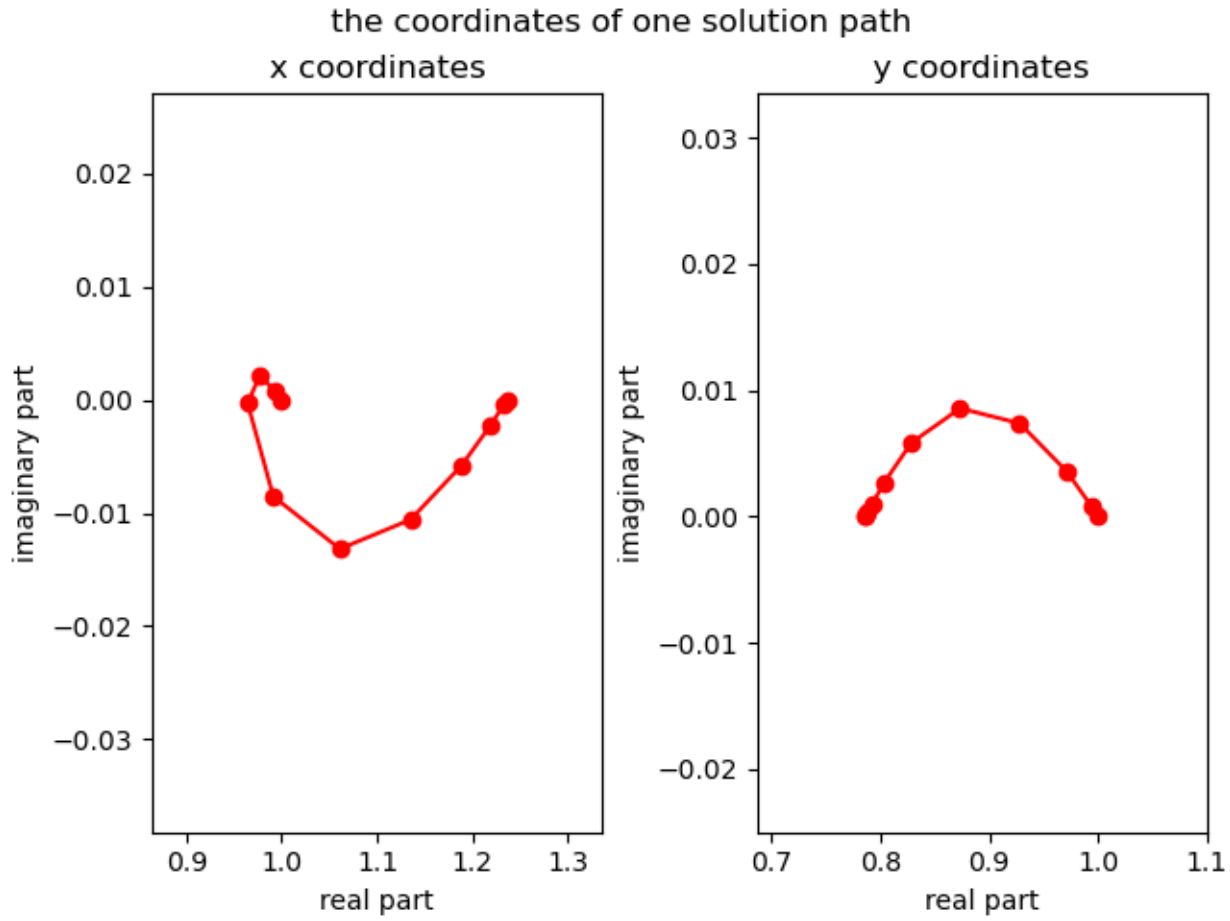


Fig. 3.1: The coordinates of one solution path.

Why do the paths in such a simple homotopy curve so much?

3.9 Increment-and-Fix Apriori Step Control

As in the previous section, an artificial-parameter homotopy is used with adaptive step size control.

In an *apriori step size control algorithm*, the predictor determines the step size for the continuation parameter t. It is called apriori because the step size is decided *before* the corrector is applied, in contrast to the aposteriori step control algorithm, where the performance of the corrector determines the step size.

Let us first define the target and start system for the running example.

```
target = ['x^2 + 4*y^2 - 4;', '2*y^2 - x;']
```

The start system is based on the total degree.

```
from phcpy.starters import total_degree_start_system
start, startsols = total_degree_start_system(target)
```

The `len(qsols)` returns `4`, equal to the total degree of the target system.

3.9.1 let the path trackers run

To run the path trackers in double precision:

```
from phcpy.curves import double_track
```

For reproducible run, the seed of the random numbers is set:

```
from phcpy.dimension import set_seed
set_seed(2024)
```

The output of

::
    gamma, sols = double_track(target, start, startsols) gamma

is

```
(-0.995051052069111-0.09936500277338699j)
```

By default, `double_track` generates a random γ and returns the generated value. This value can then be used in a second run.

The second result of `double_track` are the end points of the solution paths, printed by

```
for (idx, sol) in enumerate(sols):
    print('Solution', idx+1, ':')
    print(sol)
```

which has as output

```
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.23606797749979E+00   0.00000000000000E+00
```

(continues on next page)

(continued from previous page)

```
 y :  7.86151377757423E-01   0.00000000000000E+00
== err :  9.930E-17 = rco :  1.998E-01 = res :  1.274E-17 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.23606797749979E+00   0.00000000000000E+00
 y : -7.86151377757423E-01   0.00000000000000E+00
== err :  9.930E-17 = rco :  1.998E-01 = res :  1.274E-17 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -3.23606797749979E+00   0.00000000000000E+00
 y :  0.00000000000000E+00   1.27201964951407E+00
== err :  0.000E+00 = rco :  1.079E-01 = res :  0.000E+00 =
Solution 4 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -3.23606797749979E+00   0.00000000000000E+00
 y :  0.00000000000000E+00  -1.27201964951407E+00
== err :  0.000E+00 = rco :  1.079E-01 = res :  0.000E+00 =
```

Suppose we would want to recompute the first path in quad double precision.

```
from phcpy.curves import quad_double_track
```

Even if we track only one path, the start solution must be given in a list of one element.

```
gamma, qdsol = quad_double_track(target, start, [startsols[0]])
gamma
```

The first output is again the generated gamma constant :

```
(-0.995051052069111-0.0993650027733869 9j)
```

and the solution is printed with `print(qdsol[0])`

```
t : 1.000000000000000000000000000000000000000000000000000000000000000000000000000000E+00      0.
↪000000000000000000000000000000000000000000000000000000000000000000000000000000E+00
m : 1
the solution for t :
 x : 1.236067977499789696409173668731276235440618359611525724270897245 4E+00      6.
↪5642436311782035838428757620506362271925806587869959161667835617E-197
 y : 7.861513777574232860695585858429589295231220578377232376649019701 1E-01      7.
↪7794941850937841850195885945247107364034811096136093162063963605E-198
== err :  2.654E-67 = rco :  1.998E-01 = res :  4.175E-68 =
```

For this example, we could as well run a couple of extra steps of Newton's method at the end, but suppose that we wanted to track the paths with much smaller tolerances.

3.9.2 tuning tolerances of the path trackers

Let us redo the last run, but now with much smaller tolerances on the corrector.

```
from phcpy.curves import write_parameters
write_parameters()
```

which shows the current values of the parameters and tolerances:

```
Values of the HOMOTOPY CONTINUATION PARAMETERS :
 1. gamma : (-0.995051052069111-0.09936500277338699j)
 2. degree of numerator of Pade approximant    : 5
 3. degree of denominator of Pade approximant  : 1
 4. maximum step size                          : 0.1
 5. minimum step size                          : 1e-06
 6. multiplication factor for the pole radius  : 0.5
 7. multiplication factor for the curvature    : 0.005
 8. tolerance on the residual of the predictor : 0.001
 9. tolerance on the residual of the corrector : 1e-08
10. tolerance on zero series coefficients      : 1e-12
11. maximum number of corrector steps          : 4
12. maximum steps on a path                    : 1000
```

To set a particular value of a tolerance, we use

```
from phcpy.curves import set_parameter_value
```

To set the tolerance for the relative precision for the residuals along the path to `1.0e-32`, the parameter at position `9` has to be set, as follows:

```
set_parameter_value(9, 1.0e-32)
```

and then `write_parameters()` shows

```
Values of the HOMOTOPY CONTINUATION PARAMETERS :
 1. gamma : (-0.995051052069111-0.09936500277338699j)
 2. degree of numerator of Pade approximant    : 5
 3. degree of denominator of Pade approximant  : 1
 4. maximum step size                          : 0.1
 5. minimum step size                          : 1e-06
 6. multiplication factor for the pole radius  : 0.5
 7. multiplication factor for the curvature    : 0.005
 8. tolerance on the residual of the predictor : 0.001
 9. tolerance on the residual of the corrector : 1e-32
10. tolerance on zero series coefficients      : 1e-12
11. maximum number of corrector steps          : 4
12. maximum steps on a path                    : 1000
```

Now we rerun the first path once more.

```
gamma, qdsol = quad_double_track(target, start, [startsols[0]])
gamma
```

with value for `gamma` equal to

```
(-0.995051052069111-0.09936500277338699j)
```

and `print(qsol[0])` shows

```
t : 1.00000000000000000000000000000000000000000000000000000000000E+00      0.
↪000000000000000000000000000000000000000000000000000000000000E+00
m : 1
the solution for t :
 x : 1.23606797749978969640917366873127623544061835961152572427089724 54E+00      -5.
↪73862975392898515719315741338737601833348618881333612012303240 73E-262
 y : 7.86151377757423286069558585842958929523122057837723237664901970 11E-01      9.
↪07383641235475513045273450815712981663070306967903307489968118 80E-262
== err :  2.654E-67 = rco :  1.998E-01 = res :  4.175E-68 =
```

Observe that the small values of `err` and `res` (forward and backward error respectively) are very close to the quad double precision.

For the experiments in the next section, the values of the continuation parameters must be reset to their defaults.

```
from phcpy.curves import set_default_parameters
set_default_parameters()
write_parameters()
```

which then writes

```
Values of the HOMOTOPY CONTINUATION PARAMETERS :
 1. gamma : (-0.10983921948171692-0.9939493678571596j)
 2. degree of numerator of Pade approximant    : 5
 3. degree of denominator of Pade approximant  : 1
 4. maximum step size                          : 0.1
 5. minimum step size                          : 1e-06
 6. multiplication factor for the pole radius  : 0.5
 7. multiplication factor for the curvature    : 0.005
 8. tolerance on the residual of the predictor : 0.001
 9. tolerance on the residual of the corrector : 1e-08
10. tolerance on zero series coefficients      : 1e-12
11. maximum number of corrector steps          : 4
12. maximum steps on a path                    : 1000
```

The first parameter is the `gamma` parameter. Let us fix this parameter as well.

```
set_parameter_value(1, complex(0.977935486240886, 0.20890711991409452))
write_parameters()
```

and as confirmation, we then see:

```
Values of the HOMOTOPY CONTINUATION PARAMETERS :
 1. gamma : (0.977935486240886+0.20890711991409452j)
 2. degree of numerator of Pade approximant    : 5
 3. degree of denominator of Pade approximant  : 1
 4. maximum step size                          : 0.1
 5. minimum step size                          : 1e-06
 6. multiplication factor for the pole radius  : 0.5
 7. multiplication factor for the curvature    : 0.005
```

(continues on next page)

(continued from previous page)

```
 8. tolerance on the residual of the predictor : 0.001
 9. tolerance on the residual of the corrector : 1e-08
10. tolerance on zero series coefficients      : 1e-12
11. maximum number of corrector steps          : 4
12. maximum steps on a path                    : 1000
```

3.9.3 a step-by-step path tracker

When we run a path tracker, or let a path tracker run, then the path tracker has the control of the order of execution. In a step-by-step path tracker, we can ask the path tracker for the next point of the path, which is useful to plot the points along a path.

```
from phcpy.curves import initialize_double_artificial_homotopy
from phcpy.curves import set_double_solution, get_double_solution
from phcpy.curves import double_predict_correct
from phcpy.curves import double_t_value, double_closest_pole
```

We first initialize the artificial-parameter homotopy with the target and start system as follows:

```
initialize_double_artificial_homotopy(target, start)
```

and then set the first start solution:

```
set_double_solution(len(target), startsols[0])
```

The first predictor-corrector step is executed by

```
double_predict_correct()
pole = double_closest_pole()
pole
```

and we see the coordinates of the closest pole:

```
(0.15514554922219997, -0.08850786865995859)
```

and the next value for the continuation parameter is retrieved by

```
tval = double_t_value()
tval
```

which shows `0.089308152284921`. The corresponding point on the path is obtained by

```
nextsol = get_double_solution()
print(nextsol)
```

which prints

```
t :  8.93081522849210E-02   0.00000000000000E+00
m : 1
the solution for t :
 x :  9.95494997861267E-01   9.17030925125147E-04
 y :  9.95350349513970E-01   9.80921487331958E-04
== err :  8.233E-17 = rco :  9.210E-01 = res :  2.265E-17 =
```

To continue, run the statements

```
double_predict_correct()
pole = double_closest_pole()
print('closest pole :', pole)
nextsol = get_double_solution()
print(nextsol)"
```

and the output is

```
closest pole : (0.35111736232319435, -0.048143303085946214)
t :  1.89308152284921E-01   0.00000000000000E+00
m : 1
the solution for t :
 x :  9.79230904940275E-01   3.20751244815921E-03
 y :  9.75023847993341E-01   4.94652202413876E-03
== err :  1.146E-16 = rco :  7.493E-01 = res :  8.934E-18 =
```

To select the coordinates of the solutions, we convert to a dictionary, so we need the `strsol2dict` function.

```
from phcpy.solutions import strsol2dict
```

```
dictsol = strsol2dict(nextsol)
dictsol['t']
```

which shows `(0.189308152284921+0j)`.

In the code cell below, the loop continues calling `get_double_solution` until the value of the continuation parameter is less than 1.0. The real part and imaginary part of the gamma constant are fixed for a deterministic run.

```
initialize_double_artificial_homotopy(target, start)
set_double_solution(len(target), startsols[0])
tval = 0.0
poles = []
path = [startsols[0]]
while tval < 1.0:
    double_predict_correct()
    pole = double_closest_pole()
    locp = (tval+pole[0], pole[1])
    poles.append(locp)
    nextsol = get_double_solution()
    dictsol = strsol2dict(nextsol)
    tval = dictsol['t'].real
    path.append(nextsol)
```

To see the values of the `x`-coordinate of the points on the path:

```
for sol in path:
    print(strsol2dict(sol)['x'])
```

which prints

```
(1+0j)
(0.995494997861267+0.00091703090925125147j)
(0.979230904940275+0.00320751244815921j)
```

(continues on next page)

(continued from previous page)

```
(0.967072243680787+0.00258238975580037j)
(0.969688665229834-0.00601218730792353j)
(1.00775870703128-0.016752186008432j)
(1.07530651742827-0.0205585999286002j)
(1.14379875229794-0.0156022327531589j)
(1.19382074878865-0.00821447396701445j)
(1.22145462283616-0.00302771209969062j)
(1.23348940570526-0.000547947192401931j)
(1.23606797749979+0j)
```

For plotting, the real parts are separated from the imaginary parts, for `x` and for `y`.

```
xre = [strsol2dict(sol)['x'].real for sol in path]
xim = [strsol2dict(sol)['x'].imag for sol in path]
yre = [strsol2dict(sol)['y'].real for sol in path]
yim = [strsol2dict(sol)['y'].imag for sol in path]
```

Let us plot the coordinates of this first solution path.

::

    import matplotlib.pyplot as plt

Then Fig. 3.2 is produced by the code below:

```
fig, axs = plt.subplots(1, 2, constrained_layout=True)
fig.suptitle('the coordinates of one solution path')
axs[0].set_title('x coordinates')
axs[0].set_xlabel('real part')
axs[0].set_ylabel('imaginary part')
axs[0].set_xlim(min(xre)-0.1, max(xre)+0.1)
axs[0].set_ylim(min(xim)-0.025, max(xim)+0.025)
dots, = axs[0].plot(xre,xim,'r-')
dots, = axs[0].plot(xre,xim,'ro')
axs[1].set_title('y coordinates')
axs[1].set_xlabel('real part')
axs[1].set_ylabel('imaginary part')
axs[1].set_xlim(min(yre)-0.1, max(yre)+0.1)
axs[1].set_ylim(min(yim)-0.025, max(yim)+0.025)
dots, = axs[1].plot(yre,yim,'r-')
dots, = axs[1].plot(yre,yim,'ro')
plt.savefig('incfixaprioriﬁg1')
plt.show()
```

Why do the paths in such a simple homotopy curve so much?

Let us look at the plot of the poles.

After separating real from imaginary parts

```
repoles = [p[0] for p in poles]
impoles = [p[1] for p in poles]
```

the plot in Fig. 3.3 is made by the code below.

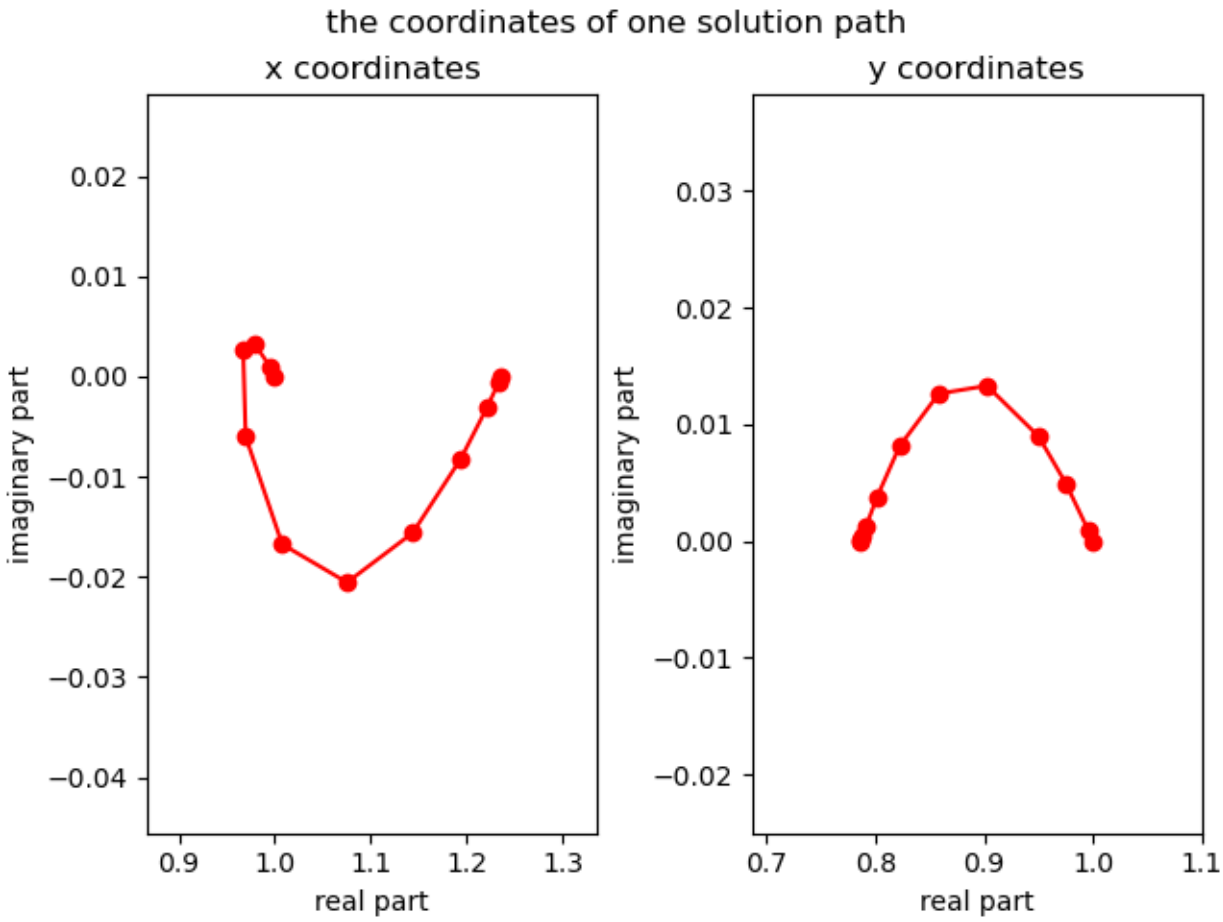


Fig. 3.2: The coordinates of one solution path.

```
fig, axs = plt.subplots(1, 1, constrained_layout=True)
axs.set_title('t goes from 0 to 1')
axs.set_xlabel('real part of poles')
axs.set_ylabel('imaginary part of poles')
dots, = axs.plot([0.0, 1.0],[0.0, 0.0],'b-')
axs.set_xlim(-0.1, 1.1)
axs.set_ylim(min(min(impoles)-0.1,-0.1), max(max(impoles)+0.1, 0.1))
dots, = axs.plot(repoles,impoles,'r+')
plt.savefig('incfixaprioriﬁg2')
plt.show()
```

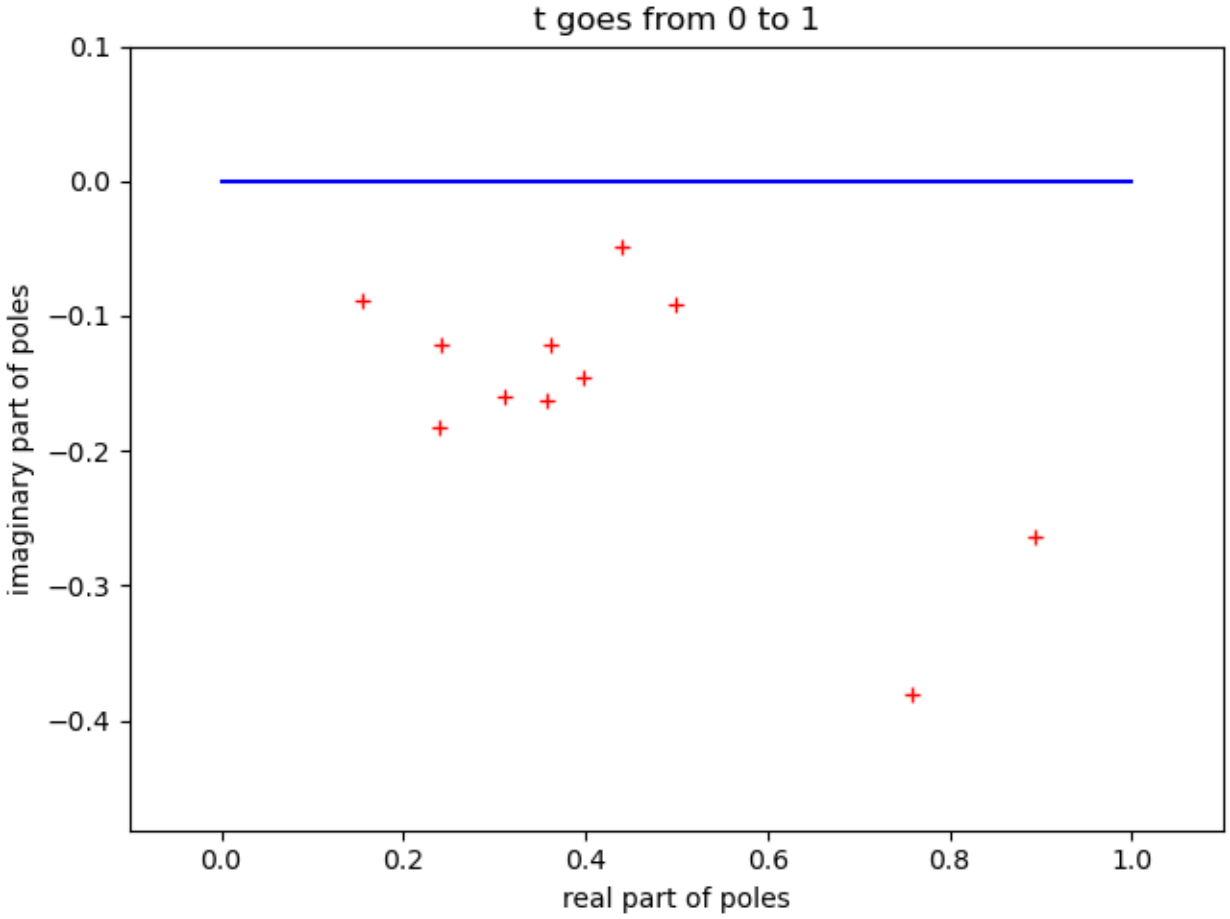


Fig. 3.3: The closest poles to one solution path.

For this plot, the closest poles appear towards the middle of the interval $[0, 1]$ where the continuation parameter t lives.

3.10 Homotopies for Problems in Enumerative Geometry

Problems in enumerative geometry have a particular structure, well suited for polynomial homotopies. Based on the Pieri root counts and the Littlewood-Richardson rule, there are homotopies which are *generical optimal* in the sense that the number of solution paths matches the number of solutions, for generic problems.

3.10.1 Pieri homotopies

A classical problem in enumerative goes as follows. Given four lines in 3-space, how many lines do meet those four given lines in a point? Although seemingly a linear problem, it turns out that the answer is two. For four given lines in general position, there are two lines meeting the four given lines.

The code below applies Pieri homotopies to compute all lines meeting four random lines in 3-space.

```
from phcpy.schubert import pieri_root_count
```

A line is represented by a matrix with 2 columns. Therefore, both `m`. the dimension of the input, and `p`, the dimension of the output, are both equal to 2.

```
pieri_root_count(2, 2)
```

returns 2. Let us compute those lines.

```
from phcpy.schubert import random_complex_matrix, run_pieri_homotopies
```

The setup for the Pieri problems are formulated for general `m`, `p` which requires `m*p` inputs for the problem to have finitely many isolated solutions.

```
(m, p) = (2, 2)
dim = m*p
L = [random_complex_matrix(m+p, m) for _ in range(dim)]
```

With the setup done, the homotopies are defined and all paths are tracked in the following code cell.

```
(fsys, fsols) = run_pieri_homotopies(m, p, 0, L)
```

The code to print the solutions

```
for (idx, sol) in enumerate(fsols):
    print('Solution', idx+1, ':')
    print(sol)
```

shows

```
Solution 1 :
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x21 : -5.90757451491546E-01  -2.73796238095519E-02
 x31 : -9.20374122953531E-01   2.18442665351047E-01
 x32 :  8.14344855671798E-01  -5.32153788006048E-01
 x42 : -5.75636214663051E-01  -2.78447850739093E-01
== err :  2.304E-15 = rco :  3.053E-02 = res :  1.388E-15 =
Solution 2 :
```

(continues on next page)

(continued from previous page)

```
t :  0.0000000000000E+00   0.0000000000000E+00
m : 1
the solution for t :
 x21 :  6.13353565245472E-01   5.36374877135208E-01
 x31 : -1.40790300814705E-01   3.49424087655845E-02
 x32 : -5.00597509424303E-01  -1.02203061067751E+00
 x42 : -1.74780414053227E+00   6.25395371456727E-01
== err :  2.558E-15 = rco :  6.714E-02 = res :  2.331E-15 =
```

The names of the variables use the indexing to denote the position in the matrix of the generators of the line. The formal root count is summarized in the poset of localization patterns.

```
from phcpy.schubert import pieri_localization_poset
poset22 = pieri_localization_poset(2, 2)
print(poset22)
```

and the poset for this problem is

```
n = 0 : ([2 3],[2 3],1)([1 4],[1 4],1)
n = 1 :
n = 2 : ([1 3],[2 4],2)
n = 3 :
n = 4 : ([1 2],[3 4],2)
```

If the degree q is nonzero, the problem of p-planes meeting m-planes is extended to computing curves of degree q that produce p-planes that meet m-planes at interpolation points. For example, consider line producing interpolating curves.

```
poset221 = pieri_localization_poset(2, 2, 1)
print(poset221)
```

```
n = 0 : ([3 4],[3 4],1)([2 5],[2 5],1)
n = 1 :
n = 2 : ([2 4],[3 5],2)
n = 3 :
n = 4 : ([2 3],[3 6],2)([2 3],[4 5],2)([1 4],[3 6],2)([1 4],[4 5],2)
n = 5 :
n = 6 : ([1 3],[4 6],8)
n = 7 :
n = 8 : ([1 2],[4 7],8)
```

There are 8 solutions as can be computed by the following code:

```
(m, p, q) = (2, 2, 1)
dim = m*p + q*(m+p)
roco = pieri_root_count(m, p, q)
```

In addition to m-planes, interpolation points must be defined.

```
L = [random_complex_matrix(m+p, m) for _ in range(dim)]
points = random_complex_matrix(dim, 1)
```

Then all paths defined by the Pieri homotopies to compute line producing curves are tracked by the statement.

```
(fsys, fsols) = run_pieri_homotopies(m, p, q, L, 0, points)
```

Then `len(fsols)` returns 8, maching the root count.

```
for (idx, sol) in enumerate(fsols):
    print('Solution', idx+1, ':')
    print(sol)
```

and the solutions are

```
Solution 1 :
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x11s0 : -2.14906949033699E-01  -6.39278480067621E-01
 x21s0 : -2.15094936097349E-01   7.13137735748634E-01
 x31s0 :  4.89419546169109E-01   7.96444688560527E-02
 x41s0 :  3.20065497988893E-01   5.85095426596230E-01
 x22s0 :  2.31759428669014E-01  -2.62654162829177E-01
 x32s0 : -9.91726822253624E-01  -6.97520762259368E-01
 x42s0 : -9.55529418833548E-01   7.00387732447402E-01
 x12s1 :  9.33661460517070E-01  -6.12105981775290E-01
== err :  0.000E+00 = rco :  0.000E+00 = res :  0.000E+00 =
Solution 2 :
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x11s0 :  1.10520066466899E+00  -3.50325368846109E-01
 x21s0 :  5.18574648064343E-01   1.32725970575267E-01
 x31s0 :  2.62487605641133E-01   3.74101095634462E-01
 x41s0 :  5.47831843633738E-01   7.39727656004381E-01
 x22s0 : -3.83564646759339E-01  -3.63155591886385E-01
 x32s0 : -4.18752363921196E-01  -1.00720592449952E-01
 x42s0 : -1.73131137487804E+00   2.32539054040540E-01
 x12s1 :  2.63582010113658E-01  -5.40149341401405E-01
== err :  0.000E+00 = rco :  0.000E+00 = res :  0.000E+00 =
Solution 3 :
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x11s0 : -1.09406342297013E+00   1.40229579344629E+00
 x21s0 : -9.14923921567758E-01  -5.69034530991378E-02
 x31s0 : -2.57253733233100E-01   1.11315272481983E+00
 x41s0 : -1.23385040712015E+00  -3.62148913829782E+00
 x22s0 : -1.48508415565715E+00  -1.46881674145667E+00
 x32s0 : -2.18057639295167E+00  -1.82823489900416E+00
 x42s0 :  3.49088194334639E+00   2.33832870569806E+00
 x12s1 : -7.99089515647370E-01   2.01242656315116E+00
== err :  0.000E+00 = rco :  0.000E+00 = res :  0.000E+00 =
Solution 4 :
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
```

(continues on next page)

(continued from previous page)

```
 x11s0 :  6.20220385308888E-01   7.14013088945019E-01
 x21s0 :  1.96622054165017E-01   6.37755854895062E-01
 x31s0 :  3.84896917178092E-01  -2.74979883047354E-01
 x41s0 :  2.85390395538486E-01   2.21952044787881E-01
 x22s0 : -3.21807655249411E-01  -5.36746050197169E-01
 x32s0 :  1.15438122582161E-01  -9.43108020255466E-01
 x42s0 : -4.47775301334196E-01  -1.25556346180423E-01
 x12s1 :  7.52973406987737E-01  -4.13342381841601E-01,
== err :  0.000E+00 = rco :  0.000E+00 = res :  0.000E+00 =
Solution 5 :
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x11s0 : -6.40605568129237E-01   5.53687431204576E-01
 x21s0 : -3.17434054891094E-01  -1.17563450907422E+00
 x31s0 : -1.27461906338159E+00   5.02576168841663E-01
 x41s0 : -1.10280584587259E+00   4.85210021219730E-01
 x22s0 :  8.42063423093745E-01   8.61583481543135E-01
 x32s0 :  6.33194409804748E-01   6.14896771183676E-01
 x42s0 :  3.64149459917250E-01   2.12619523294958E-01
 x12s1 : -2.47538557540740E-01   4.66198793734136E-02,
== err :  0.000E+00 = rco :  0.000E+00 = res :  0.000E+00 =
Solution 6 :
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x11s0 :  6.09038672836626E-01  -8.98742630676646E-01
 x21s0 :  7.42293129431968E-02   9.26153607979929E-01
 x31s0 :  8.37592049308958E-01  -4.43304301590072E-01
 x41s0 :  2.19220600974304E-02   1.14241100871086E+00,
 x22s0 :  9.44017626164027E-01  -2.71712341229773E-01
 x32s0 : -5.64425652511392E-01  -2.36806482566213E-01
 x42s0 : -7.39375051122763E-01  -4.58006540806617E-01
 x12s1 :  5.62059852022753E-01   2.93332939159073E-02
== err :  0.000E+00 = rco :  0.000E+00 = res :  0.000E+00 =
Solution 7 :
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x11s0 : -7.07534912238552E-01   4.92999096377366E-02
 x21s0 :  4.79149192690524E-02   6.23651158687302E-01
 x31s0 :  4.35563235063921E-01   2.57626463594513E-01
 x41s0 :  2.93479058252621E-01   2.14109168395354E-01
 x22s0 : -1.20085603380698E-01  -3.52342158354991E-01
 x32s0 : -1.14999164093347E+00  -3.86601319323260E-01
 x42s0 : -1.50274237644790E-01   5.36877541048399E-01
 x12s1 :  2.25543133569722E-01  -1.00744217331902E+00
== err :  0.000E+00 = rco :  0.000E+00 = res :  0.000E+00 =
Solution 8 :
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
```

(continues on next page)

(continued from previous page)

```
 x11s0 : -5.04261904420292E-02   1.59417581791122E+00
 x21s0 : -6.78725206430762E-01  -2.02091391977252E-01
 x31s0 :  4.63632342073102E-01   6.74339361230072E-01
 x41s0 :  1.10851072800672E+00   1.07992063150119E+00
 x22s0 :  9.14318142078649E-01   2.67548922924622E-01
 x32s0 :  2.66219598193448E-01   1.24867156045807E+00
 x42s0 : -2.64481349873839E+00  -2.05243627160146E-01
 x12s1 :  3.30575444135274E-01  -1.42397007492760E+00
== err :  0.000E+00 = rco :  0.000E+00 = res :  0.000E+00 =
```

The index following the `s` in the variable name represents the degree of the parameter `s` in the curve that produces lines in 3-space.

3.10.2 Littlewood-Richardson homotopies

A Schubert condition is represented by a sequence of brackets. Each bracket represents conditions on the dimensions of the intersections with the given inputs.

With Littlewood-Richardson rule, we count the number of solutions, resolving the Schubert condition.

```
from phcpy.schubert import resolve_schubert_conditions
```

The intersection conditions are defined below.

```
brackets = [[2, 4, 6], [2, 4, 6], [2, 4, 6]]
```

We are looking for 3-planes X in 6-planes that meet flags as follows:

$$\begin{aligned} 1.\dim(X \cap \langle f_1, f_2 \rangle) &= 1. \\ 2.\dim(X \cap \langle f_1, f_2, f_3, f_4 \rangle) &= 2. \\ 3.\dim(X \cap \langle f_1, f_2, f_3, f_4, f_5, f_6 \rangle) &= 3. \end{aligned}$$

For these conditions, there are finitely many solutions X. The number of solution is computed as follows.

```
roco = resolve_schubert_conditions(6, 3, brackets)
```

Littlewood-Richardson homotopies track exactly as many paths as the value of `roco`, which is 2 for this problem.

```
from phcpy.schubert import double_littlewood_richardson_homotopies as lrh
```

Tracking all paths of 3-planes in 6-space defined by Littlewood-Richardson homotopies is done by the execution of the statement.

```
(count, flags, sys, sols) = lrh(6, 3, brackets, verbose=False)
```

The value of `count` is 2 and the solutions are

```
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x11 : -1.71828539203956E+00   3.70396971521702E-01
 x32 : -9.38154978067327E-01   4.39465496011351E-01
```

(continues on next page)

(continued from previous page)

```
 x53 : -4.43650959809938E-01   9.55468566341054E-02
== err :  0.000E+00 = rco :  1.000E+00 = res :  4.785E-16 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x11 : -6.49381975027210E-01  -4.99975537206415E-01
 x32 : -1.40857387994158E+00   4.74177393405449E-01
 x53 : -7.94711695711224E-01  -1.11583537216770E-01
== err :  0.000E+00 = rco :  1.000E+00 = res :  4.785E-16 =
```

and once again, the indices of the variable names indicate the position of the numbers in the 3-planes.

3.11 Newton's Method, Deflation and Multiplicity

The instructions in the code cells in this chapter require the following functions to be imported:

```
from phcpy.solutions import make_solution, strsol2dict
from phcpy.deflation import double_newton_step
from phcpy.deflation import double_double_newton_step
from phcpy.deflation import quad_double_newton_step
from phcpy.deflation import double_deflate
from phcpy.deflation import double_multiplicity
```

3.11.1 Newton's method

Let us start with a simple system of two polynomials in two unknowns.

```
pols = ['x^2 + 4*y^2 - 4;', '2*y^2 - x;']
```

At a regular solution, Newton's method doubles the accuracy in each step. For the example, we start at a solution where the coordinates are given with only three decimal places.

```
sols = [make_solution(['x', 'y'], [1.23, -0.786])]
print(sols[0])
```

Here is then the solution with three decimal places of accuracy:

```
t : 0.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
 x : 1.23000000000000E+00  0.0
 y : -7.86000000000000E-01  0.0
== err : 0.000E+00 = rco : 1.000E+00 = res : 0.000E+00 =
```

The function `double_newton_step` expects a list of solution strings as its second argument.

```
sols = double_newton_step(pols, sols)
print(sols[0])
```

The outcome of one step with Newton's method in double precision is

```
t :  0.00000000000000E+00   0.00000000000000E+00
m : 0
the solution for t :
 x :  1.23607623318386E+00   0.00000000000000E+00
 y : -7.86154018188250E-01   0.00000000000000E+00
== err :  6.230E-03 = rco :  1.998E-01 = res :  3.706E-05 =
```

Observe the values for `err` (forward error), `rco` (estimated inverse of the condition number), and `res` (the residual or backward error).

The multiplicity field `m` turned `0` because the default tolerance was not reached and the solution could not be counted as a proper solution. Let us reset the multiplicity, making a new solution with the values from the `sols[0]`. The string representation of the solution is converted into a dictionary first:

```
sold = strsol2dict(sols[0])
sold
```

which yields

```
{'t': 0j,
 'm': 0,
 'err': 0.00623,
 'rco': 0.1998,
 'res': 3.706e-05,
 'x': (1.23607623318386+0j),
 'y': (-0.78615401818825+0j)}
```

and then we make a new solution:

```
sol = make_solution(['x', 'y'], [sold['x'], sold['y']])
print(sol)
```

As input for the next Newton step we will use

```
t : 0.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
 x : 1.23607623318386E+00  0.00000000000000E+00
 y : -7.86154018188250E-01  0.00000000000000E+00
== err : 0.000E+00 = rco : 1.000E+00 = res : 0.000E+00 =
```

Calling `double_newton_step` for the second time

```
sols = [sol]
sols = double_newton_step(pols, sols)
print(sols[0])
```

gives

```
t :  0.00000000000000E+00   0.00000000000000E+00
m : 0
the solution for t :
 x :  1.23606797751503E+00   0.00000000000000E+00
 y : -7.86151377766704E-01   0.00000000000000E+00
== err :  1.090E-05 = rco :  1.998E-01 = res :  1.100E-10 =
```

Observe that the value of `res` dropped from magnitude `1.0e-5` down to `1.0e-10`, corresponding to the well conditioning of the root. However, the multiplicy field is still zero because the estimate for the forward error is still too high.

The third application of the Newton step

```
sold = strsol2dict(sols[0])
sol = make_solution(['x', 'y'], [sold['x'], sold['y']])
sols = [sol]
sols = double_newton_step(pols, sols)
print(sols[0])
```

then yields the solution

```
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.23606797749979E+00   0.00000000000000E+00
 y : -7.86151377757423E-01   0.00000000000000E+00
== err :  2.452E-11 = rco :  1.998E-01 = res :  4.441E-16 =
```

The value for the residual `res` is very close to machine precision and the solution is considered a proper regular solution.

We can double the precision to double double, and continue with the previous approximate solution:

```
sols = double_double_newton_step(pols, sols)
print(sols[0])
```

what is printed is below

```
t : 0.00000000000000000000000000000000E+00      0.00000000000000000000000000000000E+00
m : 1
the solution for t :
 x : 1.23606797749978969640917366873130E+00      0.00000000000000000000000000000000E+00
 y : -7.86151377757423286069558585843026E-01     0.00000000000000000000000000000000E+00
== err :  3.036E-16 = rco :  1.998E-01 = res :  3.944E-31 =
```

and doing this once more

```
sols = double_double_newton_step(pols, sols)\n",
print(sols[0])"
```

shows

```
t : 0.00000000000000000000000000000000E+00      0.00000000000000000000000000000000E+00
m : 1
the solution for t :
 x : 1.23606797749978969640917366873128E+00      0.00000000000000000000000000000000E+00
 y : -7.86151377757423286069558585842966E-01     0.00000000000000000000000000000000E+00
== err :  6.272E-32 = rco :  1.998E-01 = res :  0.000E+00 =
```

and then on to quad double precision:

```
sols = quad_double_newton_step(pols, sols)
print(sols[0])
```

which then gives the solution:

```
t : 0.00000000000000000000000000000000000000000000000000000000000000000000000000000E+00      0.
↪000000000000000000000000000000000000000000000000000000000000000000000000000000E+00
m : 1
the solution for t :
 x : 1.23606797749978969640917366873127623544061835961152572427089724 54E+00      0.
↪000000000000000000000000000000000000000000000000000000000000000000000000000000E+00
 y : -7.86151377757423286069558585842958929523122057837723237664901970 15E-01     0.
↪000000000000000000000000000000000000000000000000000000000000000000000000000000E+00
== err :  7.070E-33 = rco :  1.998E-01 = res :  2.101E-64 =
```

3.11.2 deflation

At an isolated singular solution, *deflation* is a method to restore the quadratic convergence of Newton's method.

```
pols = ['(29/16)*x^3 - 2*x*y;', 'x^2 - y;']
```

Obviously, `(0, 0)` is a root of the `pols` but even if we start very close to `(0, 0)`, Newton's method converges extremely slowly.

```
sols = [make_solution(['x', 'y'],[1.0e-6, 1.0e-6])]
print(sols[0])
```

The initial solution is then

```
t : 0.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
 x : 1.00000000000000E-06  0.0
 y : 1.00000000000000E-06  0.0
== err : 0.000E+00 = rco : 1.000E+00 = res : 0.000E+00 =
```

Running one step with Newton's method:

```
sols = double_newton_step(pols, sols)
print(sols[0])
```

gives

```
t :  0.00000000000000E+00   0.00000000000000E+00
m : 0
the solution for t :
 x :  9.99999906191101E-07   0.00000000000000E+00
 y :  9.99999812409806E-13   0.00000000000000E+00
== err :  1.000E-06 = rco :  5.625E-13 = res :  1.875E-19 =
```

Observe the tiny value for `rco`, the estimate of the inverse of the condition number.

The next step with Newton's method

```
sols = double_newton_step(pols, sols)
print(sols[0])
```

shows

```
t :  0.00000000000000E+00   0.00000000000000E+00
m : 0
the solution for t :\n",
 x :  6.66666604160106E-07   0.00000000000000E+00
 y :  3.33333270859482E-13   0.00000000000000E+00
== err :  3.333E-07 = rco :  2.778E-14 = res :  1.111E-13 =
```

which does not brings us much closer to `(0, 0)`.

Now we apply deflation in double precison:

```
solsd = double_deflate(pols, sols)
print(solsd[0])
```

which then yields

```
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  9.46532112069346E-24   4.09228221015004E-24
 y :  1.02357542351685E-24  -2.03442589046821E-24
== err :  5.292E-12 = rco :  5.608E-03 = res :  1.885E-15 =
```

Deflation also works on systems with more equations than unknowns.

```
pols = ['x^2;', 'x*y;', 'y^2;']
```

Once again, `(0, 0)` is the obvious root of `pols`.

```
sols = [make_solution(['x', 'y'], [1.0e-6, 1.0e-6])]
print(sols[0]
```

So, we start at

```
t : 0.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
 x : 1.00000000000000E-06  0.0
 y : 1.00000000000000E-06  0.0
== err : 0.000E+00 = rco : 1.000E+00 = res : 0.000E+00 =
```

and apply deflation

```
sols = double_deflate(pols, sols, tolrnk=1.0e-8)
print(sols[0])
```

which shows

```
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.25000000000000E-07   0.00000000000000E+00
 y :  1.25000000000000E-07   0.00000000000000E+00
== err :  1.250E-07 = rco :  8.165E-01 = res :  1.562E-14 =
```

This is not an improvement, because the tolerance on the numerical rank, given by `tolrnk=1.0e-8` is too severe. If we relax this tolerance to `1.0e-4`,

```
sols = double_deflate(pols, sols, tolrnk=1.0e-4)
print(sols[0])
```

then we obtain

```
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -5.87747175411144E-39  -5.14278778484751E-39
 y :  2.93873587705572E-39  -1.83670992315982E-40
== err :  2.757E-23 = rco :  4.082E-01 = res :  1.984E-38 =
```

and the coordinates of this numerical approximation are well below the double precision.

3.11.3 multiplicity structure

The multiplicity can be computed *locally* starting at the solution. Consider the following example:

```
pols = [ 'x**2+y-3;', 'x+0.125*y**2-1.5;']
```

The solution `(1, 2)` is a multiple root of `pols`.

Let us prepare the input:

```
sol = make_solution(['x', 'y'], [1, 2])
print(sol)
```

so, we work with

```
t : 0.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
 x : 1.00000000000000E+00  0.0
 y : 2.00000000000000E+00  0.0
== err : 0.000E+00 = rco : 1.000E+00 = res : 0.000E+00 =
```

as input to `double_multiplicity`

```
multiplicity, hilbert_function = double_multiplicity(pols, sol)
```

The value of `multiplicity` equals `3` which is confirmed by the content of `hilbert_function`:

```
[1, 1, 1, 0, 0, 0]
```

Thus, the multiplicity of `(1, 2)` as a root of `pols` equals three.

3.12 Arc Length Parameter Continuation

With increment and fix continuation, the continuation parameter t is fixed during the corrector stage. With *arc length parameter continuation*, the parameter t is variable during the correction stage. This leads to a numerically effective algorithm to compute quadratic turning points.

In this section, the following functions are used:

```
from phcpy.solutions import make_solution
from phcpy.sweepers import double_real_sweep
from phcpy.sweepers import double_complex_sweep
```

3.12.1 computing a real quadratic turning point

Arc length parameter continuation is applied in a real sweep which ends at a quadratic turning point. The homotopy is defined below.

```
circle = ['x^2 + y^2 - 1;', 'y*(1-s) + (y-2)*s;']
```

At `s` equal to zero, `y` is zero and we have ± 1 as the solutions for `x`. The two start solutions are defined by

```
first = make_solution(['x', 'y', 's'], [1.0, 0.0, 0.0])
second = make_solution(['x', 'y', 's'], [-1.0, 0.0, 0.0])
startsols = [first, second]
```

With the start solutions defined, we then start the sweep in the real parameter space.

```
newsols = double_real_sweep(circle, startsols)
```

and print the solutions with

```
for (idx, sol) in enumerate(newsols):
    print('Solution', idx+1, ':')
    print(sol)
```

The solutions at the end of the two paths are

```
Solution 1 :
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -2.46519032881566E-32   0.00000000000000E+00
 y :  1.00000000000000E+00   0.00000000000000E+00
 s :  5.00000000000000E-01   0.00000000000000E+00
== err :  0.000E+00 = rco :  1.000E+00 = res :  0.000E+00 =
Solution 2 :
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  2.46519032881566E-32   0.00000000000000E+00
 y :  1.00000000000000E+00   0.00000000000000E+00
 s :  5.00000000000000E-01   0.00000000000000E+00
== err :  0.000E+00 = rco :  1.000E+00 = res :  0.000E+00 =
```

The homotopy stopped at `s` equal to `0.5` for at that value of `s` the solution for `(x, y)` is the double point `(0, 1)`.

At a turning point, the real paths turn back in real space. If moving forward, the solution points can continue, but then as a pair of complex conjugated paths, with nonzero imaginary parts.

3.12.2 complex parameter homotopy continuation

Sweeping the parameter space with a convex linear combination of the parameters, no singularities are encountered.

```
circle = ['x^2 + y^2 - 1;']
```

The `circle` defines a natural parameter homotopy, where `y` is the parameter, starting at zero. For `y` equal to zero, the corresponding values for `x` in the start solutions are ± 1, defined below:

```
first = make_solution(['x', 'y'], [1.0, 0.0])
second = make_solution(['x', 'y'], [-1.0, 0.0])
startsols = [first, second]
```

In a complex sweep on a natural parameter homotopy, we have to define which variable will be the continuation parameter and we have to provide start and target values, giving real and imaginary parts.

```
par = ['y']
start = [0, 0]
target = [2, 0]
```

Then we run a complex sweep in double precision as:

```
newsols = double_complex_sweep(circle, startsols, 2, par, start, target)
```

The output of

```
for (idx, sol) in enumerate(newsols):
    print('Solution', idx+1, ':')
    print(sol)
```

is

```
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  0.00000000000000E+00  -1.73205080756888E+00
 y :  2.00000000000000E+00   0.00000000000000E+00
== err :  1.282E-16 = rco :  1.000E+00 = res :  4.441E-16 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  0.00000000000000E+00   1.73205080756888E+00
 y :  2.00000000000000E+00   0.00000000000000E+00
== err :  1.282E-16 = rco :  1.000E+00 = res :  4.441E-16 =
```

At the target value for `y`, we arrived at a complex conjugated pair of solutions.

3.13 Power Series Expansions

A polynomial homotopy defines algebraic curves. With Newton's method, power series expansions of the algebraic can be computed. We distinguish between

1. Taylor series that start at a regular point; and
2. power series starting at leading terms of a series.

The functions used in this section are imported below.

```
from phcpy.solutions import make_solution
from phcpy.series import double_newton_at_point
from phcpy.series import double_newton_at_series
from phcpy.series import double_pade_approximants
```

3.13.1 Taylor series and Pade approximants

The function

$$f(z) = \sqrt{\frac{1+z/2}{1+2z}}$$

is a solution $x(s)$ of the homotopy

$$(1-s)(x^2-1) + s(3x^2 - 3/2) = 0.$$

which is defined by the `pol` below:

```
pol = ['(x^2 - 1)*(1-s) + (3*x^2 - 3/2)*s;']
```

At `s` equal to zero, the values for `x` are ± 1. The start solutions are defined as

```
variables = ['x', 's']
sol1 = make_solution(variables, [1, 0])
sol2 = make_solution(variables, [-1, 0])
sols = [sol1, sol2]
```

It is always good to double check by printing the solutions `sols`:

```
for (idx, sol) in enumerate(sols):
    print('Solution', idx+1, ':')
    print(sol)
```

which shows

```
Solution 1 :
t : 0.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
 x : 1.00000000000000E+00  0.0
 s : 0.00000000000000E+00  0.0
== err : 0.000E+00 = rco : 1.000E+00 = res : 0.000E+00 =
Solution 2 :
t : 0.00000000000000E+00 0.00000000000000E+00
```

(continues on next page)

(continued from previous page)

```
m : 1
the solution for t :
 x : -1.00000000000000E+00  0.0
 s : 0.00000000000000E+00  0.0
== err : 0.000E+00 = rco : 1.000E+00 = res : 0.000E+00 =
```

In calling `double_newton_at_point` we must declare the index of the variable which serves as the parameter. As the parameter `s` is the second parameter, we set `idx = 2`. Other parameters are the degree of the series, by default `maxdeg=4` and the number of Newton steps, by default `nbr=4`.

```
srs = double_newton_at_point(pol, sols, idx=2)
srs
```

which returns

```
[[' + 3.69287109375000E+00*s^4 - 2.08593750000000E+00*s^3 + 1.21875000000000E+00*s^2 - 7.
↪50000000000000E-01*s + 1;'],
 [' - 3.69287109375000E+00*s^4 + 2.08593750000000E+00*s^3 - 1.21875000000000E+00*s^2 + 7.
↪50000000000000E-01*s - 1;']]
```

The power series are polynomials which should be read from right to left.

Pade approximants are rational expressions which agree with the power series expansions.

```
pad = double_pade_approximants(pol, sols, idx=2)
pad
```

which shows

```
[['(9.53124999999999E-01*s^2 + 2.12500000000000E+00*s + 1)/(1.89062500000000E+00*s^2 + 2.
↪87500000000000E+00*s + 1)'],
 ['( - 9.53124999999999E-01*s^2 - 2.12500000000000E+00*s - 1)/(1.89062500000000E+00*s^2␣
↪+ 2.87500000000000E+00*s + 1)']]
```

To evaluate the first Pade approximant at `0.1` as the value for `s`, we first replace the string `"s"` by the string `"0.1"`.

```
p0 = pad[0][0].replace('s', '0.1')
p0
```

which leads to the string

```
'(9.53124999999999E-01*0.1^2 + 2.12500000000000E+00*0.1 + 1)/(1.89062500000000E+00*0.1^2␣
↪+ 2.87500000000000E+00*0.1 + 1)'
```

Observe the `^` which must be replaced by `**`, as done by

```
p1 = p0.replace('^', '**')
p1
```

and then we have the expression in the string `p1`:

```
'(9.53124999999999E-01*0.1**2 + 2.12500000000000E+00*0.1 + 1)/(1.89062500000000E+00*0.
↪1**2 + 2.87500000000000E+00*0.1 + 1)'
```

Its numerical value, obtained via

```
ep1 = eval(p1)
ep1
```

shows

```
0.9354144241119484
```

Let us compare this now to the function

$$f(z) = \sqrt{\frac{1+z/2}{1+2z}}$$

evaluated at $z = 0.1$. The statements

```
from math import sqrt
ef1 = sqrt((1 + 0.1/2)/(1+2*0.1))
ef1
```

produce the value

```
0.9354143466934854
```

and we obtain `7.741846297371069e-08` as the outcome of the statement `abs(ep1 - ef1)`. The error shows we have about 8 decimal places correct for the value of $f(z)$ at $z = 0.1$.

3.13.2 expansions starting at series

Starting the power series expansions at a series allows to start at a singular solution, as illustrated by the Viviani curve, defines as the intersection of a sphere with a quadratic cylinder.

```
pols = [ '2*t^2 - x;', \
         'x^2 + y^2 + z^2 - 4;' , \
         '(x-1)^2 + y^2 - 1;']
```

The series starts at $x = 2t^2$.

```
lser = [ '2*t^2;', '2*t;', '2;']
```

Then Newton's method is executed, here using the default `idx=1` as `t` is the first parameter, asking for a series truncated at degree 12, using no more than 8 iterations.

```
nser = double_newton_at_series(pols, lser, maxdeg=12, nbr=8)
```

To print the expansions in `nser`, the names of the variables are used:

```
variables = ['x', 'y', 'z']
for (var, pol) in zip(variables, nser):
    print(var, '=', pol)
```

```
x = 2*t^2;
y =  - 5.46875000000000E-02*t^11 - 7.81250000000000E-02*t^9 - 1.25000000000000E-01*t^7 -␣
↪2.50000000000000E-01*t^5 - t^3 + 2*t;
z =  - 4.10156250000000E-02*t^12 - 5.46875000000000E-02*t^10 - 7.81250000000000E-02*t^8 -
↪ 1.25000000000000E-01*t^6 - 2.50000000000000E-01*t^4 - t^2 + 2;
```

The coefficients of the power series expansions indicate how fast the solutions change once we move away from the singularity.

The example below compares the series expansions at two solutions for the problem of Apollonius."

```
pols = [ 'x1^2 + 3*x2^2 - r^2 - 2*r - 1;', \
         'x1^2 + 3*x2^2 - r^2 - 4*x1 - 2*r + 3;', \
       '3*t^2 + x1^2 - 6*t*x2 + 3*x2^2 - r^2 + 6*t - 2*x1 - 6*x2 + 2*r + 3;']
```

The `pols` define an instance where the input circles are mutually touching each other. Once we start moving the input circles apart from each other, how fast do the touching circles grow? The developments start at the following terms:

```
lser1 = ['1;', '1 + 0.536*t;', '1 + 0.904*t;']
lser2 = ['1;', '1 + 7.464*t;', '1 + 11.196*t;']
```

We run Newton's method twice:

```
nser1 = double_newton_at_series(pols, lser1, idx=4, nbr=7)
nser2 = double_newton_at_series(pols, lser2, idx=4, nbr=7)
```

The statements

```
variables = ['x', 'y', 'z']
print('the first solution series :')
for (var, pol) in zip(variables, nser1):
    print(var, '=', pol)
print('the second solution series :')
for (var, pol) in zip(variables, nser2):
    print(var, '=', pol)
```

have as output

```
the first solution series :
x =  - 4.03896783473158E-28*t^4 - 4.62223186652937E-33*t + 1;
y =  - 7.73216421430735E-03*t^4 + 7.73216421430735E-03*t^3 - 1.66604983954048E-02*t^2 +␣
↪5.35898384862246E-01*t + 1;
z =  - 1.33925012716462E-02*t^3 + 2.88568297002609E-02*t^2 + 8.03847577293368E-01*t + 1;
the second solution series :
x = 1.00974195868290E-28*t^3 + 1.97215226305253E-31*t + 1;
y =  - 2.90992267835785E+02*t^4 + 2.90992267835785E+02*t^3 + 4.50166604983953E+01*t^2 +␣
↪7.46410161513775E+00*t + 1.00000000000000E+00;
z = 5.04013392501271E+02*t^3 + 7.79711431702996E+01*t^2 + 1.11961524227066E+01*t + 1.
↪00000000000000E+00;
```

Observe the difference in magnitudes of the coefficients of the series expansions, indicating that one solution will change more than the other, as we move away from the singularity.

3.14 Positive Dimensional Solution Sets

A more complete solver returns a numerical irreducible decomposition, with not only the isolated solutions, but the dimension and the degrees of all positive dimensional solution sets, for all dimensions.

3.14.1 witness sets

A witness set of a pure dimensional solution set consists of

1. the original polynomial system augmented with as many random hyperplanes as the dimension of the solution set; and
2. as many solutions to the augmented system as the degree of the solution set.

Because of the random coefficients in the hyperplanes, the solutions are generic points.

Via an embedding of the augmented system, the computation of generic points is reduced to the computation of isolated solutions, which can be handled well by the blackbox solver.

```
from phcpy.sets import double_embed
from phcpy.solver import solve
```

Let us make a witness set of the twisted cubic.

```
twisted = ['x^2 - y;', 'x^3 - z;']
```

The twisted cubic is the space curve with parametric form (t, t^2, t^3).

The dimension is 1 and we are in dimension 3. The embedded system is constructed as:

```
embtwist = double_embed(3, 1, twisted)
for pol in embtwist:
    print(pol)
```

and the embedded polynomials are

```
x^2 - y + (5.62891189304868E-01-8.26531009099448E-01*i)*zz1;
x^3 - z + (-9.84048066692442E-01-1.77902789294791E-01*i)*zz1;
zz1;
 + (8.39559929055672E-01-5.43267084889224E-01*i)*x + (-1.14034198908312E-01-9.
↪93476824832537E-01*i)*y + (-9.45117489397468E-01-3.26730670790218E-01*i)*z + (4.
↪57472148097901E-01-8.89223950259265E-01*i)*zz1+(-9.21723254199187E-01-3.
↪87848221174806E-01*i);
```

The symbol zz1 is used for the slack variable.

```
sols = solve(embtwist)
for (idx, sol) in enumerate(sols):
    print('Solution', idx+1, ':')
    print(sol)
```

and the generic points on the twisted cubic are

```
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
```

(continues on next page)

(continued from previous page)

```
m : 1
the solution for t :
 x :  6.31159601615322E-01  -1.19854989224432E+00
 y : -1.03815940148766E+00  -1.51295254501003E+00
 zz1 : -4.04959151575673E-33   1.16188546226650E-32
 z : -2.46859338404870E+00   2.89371313214052E-01
== err :  6.130E-16 = rco :  2.728E-02 = res :  3.331E-16 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -1.34539355262783E+00  -1.68943360753041E-01
 y :  1.78154195231000E+00   4.54590616632837E-01
 zz1 :  0.00000000000000E+00   0.00000000000000E+00
 z : -2.32007498983311E+00  -9.12582969448712E-01
== err :  7.943E-16 = rco :  3.560E-02 = res :  1.166E-15 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  2.81858885842759E-01   4.65799400839404E-01
 y : -1.37524650293826E-01   2.62579400293639E-01
 zz1 :  5.64466889738308E-34  -9.64221135309633E-34
 z : -1.61071872037280E-01   9.95143750451212E-03
== err :  5.737E-17 = rco :  9.995E-02 = res :  4.163E-17 =
```

As the degree of the twisted cubic is three, we have three generic points. Observe the tiny values for the slack variable zz1, as the generic points must satisfy the added random hyperplane of the augmented system.

3.14.2 homotopy membership test

A witness set can be used to decide whether any point belongs to the algebraic set represented by the witness set. The homotopy membership test is illustrated via the solution set of the cyclic 4-roots system.

```
from phcpy.families import cyclic
c4 = cyclic(4)
for pol in c4:
    print(pol)
```

Here are the four polynomials:

```
x0 + x1 + x2 + x3;
x0*x1 + x1*x2 + x2*x3 + x3*x0;
x0*x1*x2 + x1*x2*x3 + x2*x3*x0 + x3*x0*x1;
x0*x1*x2*x3 - 1;
```

Although we have four equations and four unknowns (we expect thus only isolated solutions), we know that for this particular system, the solution set is pure dimensional, of dimension 1.

To construct a witness set, we import the following functions:

```
from phcpy.sets import double_embed
from phcpy.solver import solve
from phcpy.solutions import filter_zero_coordinates as filter
```

and then execute

```
c4e1 = double_embed(4, 1, c4)
sols = solve(c4e1)
genpts = filter(sols, 'zz1', 1.0e-8, 'select')
print('generic points on the cyclic 4-roots set :')
for (idx, sol) in enumerate(genpts):
    print('Solution', idx+1, ':')
    print(sol)
```

to see the generic points on the solution curve of the cyclic 4-roots:

```
generic points on the cyclic 4-roots set :
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x0 :  9.65599349076935E-01  -1.16989010460731E+00
 x1 : -4.19638798339057E-01  -5.08421301397389E-01
 x2 : -9.65599349076935E-01   1.16989010460732E+00
 x3 :  4.19638798339057E-01   5.08421301397389E-01
 zz1 :  1.76873803944398E-16  -1.05541954650188E-16
== err :  1.859E-15 = rco :  4.629E-02 = res :  9.649E-16 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x0 : -4.72839263499989E-01  -1.41379607496008E+00
 x1 : -2.12761000919484E-01   6.36158397206689E-01
 x2 :  4.72839263499989E-01   1.41379607496008E+00
 x3 :  2.12761000919484E-01  -6.36158397206689E-01
 zz1 : -8.45579970922059E-17   3.34398025784039E-17
== err :  1.268E-15 = rco :  5.880E-02 = res :  5.892E-16 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x0 : -7.78676715642733E-01   2.78291443186817E-01
 x1 :  1.13877660574983E+00   4.06987622353548E-01
 x2 :  7.78676715642734E-01  -2.78291443186816E-01
 x3 : -1.13877660574983E+00  -4.06987622353548E-01
 zz1 : -8.30236390890514E-17  -4.21955685140459E-17
== err :  1.296E-15 = rco :  1.051E-01 = res :  1.496E-15 =
Solution 4 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x0 :  6.59761896934191E-01   5.22197413539580E-01
 x1 :  9.31898808330260E-01  -7.37592076250530E-01
```

(continues on next page)

(continued from previous page)

```
 x2 : -6.59761896934191E-01  -5.22197413539580E-01
 x3 : -9.31898808330260E-01   7.37592076250530E-01
 zz1 : -6.83231299973127E-17   7.95281480444320E-17
== err :  1.022E-15 = rco :  6.324E-02 = res :  9.147E-16 =
```

For the membership test in double precision, we use the function:

```
from phcpy.sets import double_membertest
```

Consider two test points `pt0` and `pt1`.

The first test

```
pt0 = [1, 0, -1, 0, 1, 0, -1, 0]
ismbr = double_membertest(c4e1, sols, 1, pt0)
print('Is', pt0, 'a member?', ismbr)
```

gives as output

```
Is [1, 0, -1, 0, 1, 0, -1, 0] a member? False
```

and the second test

```
pt1 = [1, 0, 1, 0, -1, 0, -1, 0]
ismbr = double_membertest(c4e1, sols, 1, pt1)
print('Is', pt1, 'a member?', ismbr)
```

yields

```
Is [1, 0, 1, 0, -1, 0, -1, 0] a member? True
```

3.14.3 monodromy breakup

The factorization into irreducible components is illustrated on a cubic curve.

```
cubic = '(x+1)*(x^2 + y^2 + 1);'
```

The input to the factorization function is a witness set.

```
from phcpy.sets import double_hypersurface_set
from phcpy.factor import double_monodromy_breakup, write_factorization
```

The construction of the witness set happens via

```
(wit, pts) = double_hypersurface_set(2, cubic)
for pol in wit:
    print(pol)\n",
print('number of witness points :', len(pts))
```

and the output is

```
x^3 + x*y^2 + x^2 + y^2 + x + (5.56101869358167E-01-8.31114138308544E-01*i)*zz1 + 1;
zz1;
 + (9.85343874390340E-01 + 1.70579744405467E-01*i)*x + y + zz1+(-1.26905195457699E+00 +␣
↪1.50366546205483E+00*i);
number of witness points : 3
```

To see the witness points, execute

```
for (idx, sol) in enumerate(pts):
    print('Solution', idx+1, ':')
    print(sol)
```

which then prints

```
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -4.30394583533542E-02  -1.45862430717631E+00
 y :  1.06264885972081E+00  -5.90772761365393E-02
 zz1 :  0.00000000000000E+00   0.00000000000000E+00
== err :  0.000E+00 = rco :  1.000E+00 = res :  0.000E+00 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.33096744731273E+00  -6.74068593392326E-02
 y : -5.39069114828172E-02  -1.66428261308763E+00
 zz1 :  0.00000000000000E+00   0.00000000000000E+00
== err :  0.000E+00 = rco :  1.000E+00 = res :  0.000E+00 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -1.00000000000000E+00   1.11022302462516E-16
 y :  2.25439582896733E+00  -1.33308571764937E+00
 zz1 :  0.00000000000000E+00   0.00000000000000E+00
== err :  0.000E+00 = rco :  1.000E+00 = res :  0.000E+00 =
```

The factorization is then computed via

```
deco = double_monodromy_breakup(wit, pts, dim=1)
```

To see the grouping of the generic points according to the irreducible factors, we do:

```
write_factorization(deco)
```

which prints

```
factor 1 : ([1, 2], 3.683680191092806e-15)
factor 2 : ([3], 9.742207041085749e-15)
```

3.14.4 cascade of homotopies

A cascade of homotopies computes generic points on all positive components of the solution set, for all dimensions.

```
pol1 = '(x^2 + y^2 + z^2 - 1)*(y - x^2)*(x - 0.5);'
pol2 = '(x^2 + y^2 + z^2 - 1)*(z - x^3)*(y - 0.5);'
pol3 = '(x^2 + y^2 + z^2 - 1)*(z - x*y)*(z - 0.5);'
pols = [pol1, pol2, pol3]"
```

The solution set of `pols` contains the sphere, the twisted cubic, some lines, and an isolated point.

To run a cascade of homotopies, import the following functions:

```
from phcpy.cascades import double_top_cascade, double_cascade_filter
```

We start at the top dimension of the solution set:

```
(embpols, sols0, sols1) = double_top_cascade(3, 2, pols)
print('at dimension 2, degree :', len(sols0))"
```

to find two witness points on the sphere:

```
at dimension 2, degree : 2
```

and then continue to the one dimensional solution sets:

```
(wp1, ws0, ws1) = double_cascade_filter(2, embpols, sols1, tol=1.0e-8)
print('at dimension 1, candidate generic points :', len(ws0))
```

to obtain:

```
at dimension 1, candidate generic points : 9
```

and then continue to the isolated solutions:

```
(wp0, ws0, ws1) = double_cascade_filter(1, wp1, ws1, tol=1.0e-8)
print('candidate isolated points :', len(ws0))
```

to find

```
candidate isolated points : 24
```

The output of the cascade needs further processing. The `solve` in the next section does all steps.

3.14.5 numerical irreducible decomposition

The computation of a numerical irreducible decomposition starts by solving the top dimensional system in an embedding and then applies a cascade of homotopies to compute candidate generic points on each positive dimensional solution set, ending at the isolated solutions. After each step in the cascade, the candidate generic points are filtered as some may lie on higher dimensional sets, and the generic points are grouped according to their irreducible factors.

```
from phcpy.decomposition import solve, write_decomposition
```

The second blackbox solver is illustrated on the following example:

```
pol0 = '(x1-1)*(x1-2)*(x1-3)*(x1-4);'
pol1 = '(x1-1)*(x2-1)*(x2-2)*(x2-3);'
pol2 = '(x1-1)*(x1-2)*(x3-1)*(x3-2);'
pol3 = '(x1-1)*(x2-1)*(x3-1)*(x4-1);'
pols = [pol0, pol1, pol2, pol3]
```

For this small example, we ask the `solve` to be silent:

```
deco = solve(pols, verbose=False)
```

and then write the decomposition in `deco` as

```
write_decomposition(deco)
```

The output is rather extensive …

```
set of dimension 0 has degree 4
the polynomials :
 + x1^4 - 10*x1^3 + 35*x1^2 - 50*x1 + 24;
x1*x2^3 - 6*x1*x2^2 - x2^3 + 11*x1*x2 + 6*x2^2 - 6*x1 - 11*x2 + 6;
x1^2*x3^2 - 3*x1^2*x3 - 3*x1*x3^2 + 2*x1^2 + 9*x1*x3 + 2*x3^2 - 6*x1 - 6*x3 + 4;
x1*x2*x3*x4 - x1*x2*x3 - x1*x2*x4 - x1*x3*x4 - x2*x3*x4
+ x1*x2 + x1*x3 + x1*x4 + x2*x3 + x2*x4 + x3*x4 - x1 - x2 - x3 - x4 + 1;
the generic points :
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x1 :  4.00000000000000E+00   0.00000000000000E+00
 x2 :  3.00000000000000E+00   0.00000000000000E+00
 x3 :  2.00000000000000E+00   0.00000000000000E+00
 x4 :  1.00000000000000E+00   0.00000000000000E+00
== err :  4.956E-26 = rco :  1.359E-01 = res :  0.000E+00 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x1 :  4.00000000000000E+00   3.85357077689325E-45
 x2 :  2.00000000000000E+00  -2.94004118110142E-50
 x3 :  2.00000000000000E+00   1.64214663788065E-46
 x4 :  1.00000000000000E+00   1.77899219103737E-47
== err :  7.105E-15 = rco :  1.191E-01 = res :  2.416E-44 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x1 :  3.00000000000000E+00   4.27642353614751E-50
 x2 :  3.00000000000000E+00   0.00000000000000E+00
 x3 :  2.00000000000000E+00   3.20731765211063E-50
 x4 :  1.00000000000000E+00   0.00000000000000E+00
== err :  1.204E-34 = rco :  1.043E-01 = res :  1.497E-49 =
Solution 4 :
t :  1.00000000000000E+00   0.00000000000000E+00
```

(continues on next page)

(continued from previous page)

```
m : 1
the solution for t :
 x1 :  2.99999999999999E+00   3.50324616081204E-45
 x2 :  2.00000000000000E+00   2.83543986140725E-45
 x3 :  2.00000000000000E+00  -7.18165462966469E-45
 x4 :  1.00000000000000E+00   6.30584308946168E-45
== err :  9.770E-15 = rco :  9.066E-02 = res :  1.066E-14 =
set of dimension 1 has degree 12
the polynomials :
 + x1^4 - 10*x1^3 + 35*x1^2 - 50*x1
+ (9.98263256449285E-01-5.89107021114901E-02*i)*zz1 + 24;
x1*x2^3 - 6*x1*x2^2 - x2^3 + 11*x1*x2 + 6*x2^2 - 6*x1 - 11*x2
+ (8.25826300922299E-02-9.96584220829855E-01*i)*zz1 + 6;
x1^2*x3^2 - 3*x1^2*x3 - 3*x1*x3^2 + 2*x1^2 + 9*x1*x3 + 2*x3^2 - 6*x1
- 6*x3 + (1.55033404443160E-01-9.87909228374127E-01*i)*zz1 + 4;
x1*x2*x3*x4 - x1*x2*x3 - x1*x2*x4 - x1*x3*x4 - x2*x3*x4
+ x1*x2 + x1*x3 + x1*x4 + x2*x3 + x2*x4 + x3*x4 - x1 - x2 - x3 - x4
+ (-8.97158012682676E-01-4.41709746642829E-01*i)*zz1 + 1;
+ (-3.53404925407865E-01-1.02449352102111E-02*i)*x1
+ (-3.23378971672498E-01-1.42919700111768E-01*i)*x2
+ (2.20216662912369E-01-2.76594687902244E-01*i)*x3
+ (8.35397253138817E-02-3.43542012415485E-01*i)*x4
+ (-2.67791727313329E-01-2.30841050904175E-01*i)*zz1 - 3.53553390593274E-01;
the generic points :
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x1 :  2.00000000000000E+00   1.66274442058684E-16
 x2 :  2.00000000000000E+00   1.93730374623830E-15
 x3 :  1.42230996999871E+00   4.73749193868512E+00
 x4 :  1.00000000000000E+00   3.09283683575416E-16
 zz1 : -1.98123724512471E-15  -4.50046576766827E-16
== err :  2.605E-14 = rco :  1.263E-02 = res :  6.855E-14 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x1 :  4.00000000000001E+00   1.93485385394018E-17
 x2 :  1.00000000000000E+00  -5.10382551856806E-18
 x3 :  2.00000000000000E+00  -3.61201796027518E-18
 x4 : -9.22964074590540E-01   5.02769047428598E+00
 zz1 : -4.05629881527961E-17  -1.18686954150410E-16
== err :  2.775E-14 = rco :  1.246E-02 = res :  7.817E-14 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x1 :  2.99999999999999E+00   3.50163623105219E-16
 x2 :  3.00000000000000E+00   8.96756225629632E-19
 x3 :  1.00000000000000E+00   2.40219551951397E-17
 x4 : -5.76987365375915E-01   6.43848410091975E+00
```

(continues on next page)

(continued from previous page)

```
 zz1 :  6.20367476284742E-17   7.05206637649815E-16
== err :  3.038E-14 = rco :  7.888E-03 = res :  3.206E-14 =
Solution 4 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x1 :  3.00000000000000E+00   1.29780826664201E-16
 x2 :  1.00000000000000E+00   9.36913548588266E-18
 x3 :  2.00000000000000E+00   8.93428893566725E-18
 x4 : -1.13099435246225E+00   4.04956808752212E+00
 zz1 :  5.94418462635302E-17   2.63521082767564E-16
== err :  2.920E-15 = rco :  2.162E-02 = res :  2.671E-16 =
Solution 5 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :\n",
 x1 :  2.00000000000000E+00  -3.65568752932322E-16
 x2 :  3.00000000000000E+00   2.82574614718779E-15
 x3 :  1.67577083520075E+00   5.70483758002075E+00
 x4 :  1.00000000000000E+00  -3.72113707233408E-16
 zz1 :  5.75972032456864E-15   1.07230899989073E-15
== err :  2.494E-14 = rco :  8.760E-03 = res :  2.468E-14 =
Solution 6 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x1 :  2.00000000000000E+00   1.76757984029786E-16
 x2 :  3.00000000000000E+00  -1.31917536806969E-17
 x3 :  1.00000000000000E+00  -4.99155351560570E-17
 x4 : -7.85017643247620E-01   5.46036171415591E+00
 zz1 : -5.60935500844177E-17  -3.57441262286022E-16
== err :  2.215E-14 = rco :  1.929E-02 = res :  1.493E-14 =
Solution 7 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x1 :  4.00000000000000E+00  -2.92505382506395E-18
 x2 :  2.00000000000000E+00  -1.74547448231340E-18
 x3 :  1.00000000000000E+00  -6.45889285731916E-19
 x4 : -1.92285640108937E-01   6.43233660615961E+00
 zz1 :  6.74420113461971E-18   1.79788532318047E-17
== err :  1.018E-14 = rco :  1.020E-02 = res :  6.776E-14 =
Solution 8 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x1 :  2.99999999999999E+00   1.21726975202562E-16
 x2 :  1.00000000000000E+00  -5.19731533632551E-18
 x3 :  1.00000000000000E+00   1.92249486589598E-17
 x4 : -2.23644470585381E-01   4.46994433787176E+00
 zz1 : -6.54642209058018E-19   2.43838870558812E-16
== err :  2.701E-14 = rco :  3.711E-02 = res :  1.071E-14 =
```

(continues on next page)

(continued from previous page)

```
Solution 9 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x1 :  3.00000000000001E+00   7.59453204499381E-16
 x2 :  2.00000000000000E+00  -3.13045199988346E-17
 x3 :  1.00000000000000E+00   2.50434205156814E-17
 x4 : -4.00315917980645E-01   5.45421421939577E+00
 zz1 :  1.89836152203951E-16   1.53275178679189E-15
== err :  6.146E-14 = rco :  1.385E-02 = res :  3.244E-14 =
Solution 10 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x1 :  4.00000000000000E+00  -1.25836606607375E-17
 x2 :  3.00000000000000E+00   1.36281411894719E-17
 x3 :  1.00000000000000E+00  -1.25232978409423E-17
 x4 : -3.68957087504202E-01   7.41660648768361E+00
 zz1 :  8.87505203002171E-17   8.08707712184250E-17
== err :  1.058E-14 = rco :  5.769E-03 = res :  6.754E-14 =
Solution 11 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x1 :  4.00000000000000E+00  -4.37056700414583E-17
 x2 :  1.00000000000000E+00   2.28708329195796E-17
 x3 :  1.00000000000000E+00  -1.93529333821221E-17
 x4 : -1.56141927136710E-02   5.44806672463562E+00
 zz1 :  1.60246975171774E-16   2.72146931495479E-16
== err :  8.019E-16 = rco :  5.425E-02 = res :  2.567E-16 =
Solution 12 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x1 :  2.00000000000000E+00   3.54994546241256E-16
 x2 :  2.00000000000000E+00   5.29877124103883E-17
 x3 :  1.00000000000000E+00   8.46409944528661E-17
 x4 : -6.08346195852355E-01   4.47609183263191E+00
 zz1 : -1.12656321968022E-16  -7.17872515969298E-16
== err :  6.181E-15 = rco :  3.233E-02 = res :  1.200E-14 =
set of dimension 2 has degree 1
the polynomials :
x1^4 - 10*x1^3 + 35*x1^2 - 50*x1
+ (9.98263256449285E-01-5.89107021114901E-02*i)*zz1
+ (-4.67731979750726E-01 + 8.83870349722439E-01*i)*zz2 + 24;
x1*x2^3 - 6*x1*x2^2 - x2^3 + 11*x1*x2 + 6*x2^2 - 6*x1 - 11*x2
+ (8.25826300922299E-02-9.96584220829855E-01*i)*zz1
+ (-8.21936276409882E-01 + 5.69579456723519E-01*i)*zz2 + 6;
+ x1^2*x3^2 - 3*x1^2*x3 - 3*x1*x3^2 + 2*x1^2 + 9*x1*x3 + 2*x3^2 - 6*x1 - 6*x3
+ (1.55033404443160E-01-9.87909228374127E-01*i)*zz1
+ (-6.80008285278479E-01-7.33204427122902E-01*i)*zz2 + 4;
x1*x2*x3*x4 - x1*x2*x3 - x1*x2*x4 - x1*x3*x4 - x2*x3*x4
```

(continues on next page)

(continued from previous page)

```
+ x1*x2 + x1*x3 + x1*x4 + x2*x3 + x2*x4 + x3*x4
- x1 - x2 - x3 - x4 + (-8.97158012682676E-01-4.41709746642829E-01*i)*zz1
+ (-2.11981526746876E-01 + 9.77273673193985E-01*i)*zz2 + 1;
+ (-3.53404925407865E-01-1.02449352102111E-02*i)*x1
+ (-3.23378971672498E-01-1.42919700111768E-01*i)*x2
+ (2.20216662912369E-01-2.76594687902244E-01*i)*x3
+ (8.35397253138817E-02-3.43542012415485E-01*i)*x4
+ (-2.67791727313329E-01-2.30841050904175E-01*i)*zz1
+ (3.28014183926858E-01-1.31934435015266E-01*i)*zz2 - 3.53553390593274E-01;
+ (-4.95361781887585E-01 + 5.82835865995319E-03*i)*x1
+ (1.04868230499720E-01 + 2.82848378492330E-01*i)*x2i
+ (-1.92957771968200E-01 + 1.72592823677708E-01*i)*x3
+ (1.50836540788987E-01-4.65440876777183E-01*i)*x4
+ (2.59311098918677E-01-3.82322947183237E-02*i)*zz1
+ (-2.61338053548032E-01-1.91269889240060E-01*i)*zz2
+ (2.54532009330191E-01 + 1.49441343387202E-02*i);
the generic points :
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x1 :  2.00000000000000E+00  -1.02870981636522E-15
 x2 :  1.00000000000000E+00  -6.89875692784332E-17
 x3 :  5.41257790254410E-01   1.83800265575193E+00
 x4 :  7.57217605925192E-01   2.01695478853197E+00
 zz1 : -1.53177350208010E-16   1.50858780682761E-15
 zz2 :  1.11064312344268E-15   9.39077236860072E-16
== err :  4.130E-14 = rco :  4.252E-02 = res :  6.734E-15 =
set of dimension 3 has degree 1
the polynomials :
  x1^4 - 10*x1^3 + 35*x1^2 - 50*x1
+ (9.98263256449285E-01-5.89107021114901E-02*i)*zz1
+ (-4.67731979750726E-01 + 8.83870349722439E-01*i)*zz2
+ (-3.46149707492286E-01-9.38179289903057E-01*i)*zz3 + 24;
 + x1*x2^3 - 6*x1*x2^2 - x2^3 + 11*x1*x2 + 6*x2^2 - 6*x1 - 11*x2
+ (8.25826300922299E-02-9.96584220829855E-01*i)*zz1
+ (-8.21936276409882E-01 + 5.69579456723519E-01*i)*zz2
+ (-9.94974897178992E-01-1.00124692177572E-01*i)*zz3 + 6;
 + x1^2*x3^2 - 3*x1^2*x3 - 3*x1*x3^2 + 2*x1^2 + 9*x1*x3 + 2*x3^2
- 6*x1 - 6*x3 + (1.55033404443160E-01-9.87909228374127E-01*i)*zz1
+ (-6.80008285278479E-01-7.33204427122902E-01*i)*zz2
+ (9.94504055917016E-01 + 1.04698055209280E-01*i)*zz3 + 4;
  x1*x2*x3*x4 - x1*x2*x3 - x1*x2*x4 - x1*x3*x4 - x2*x3*x4
+ x1*x2 + x1*x3 + x1*x4 + x2*x3 + x2*x4 + x3*x4 - x1 - x2 - x3 - x4
+ (-8.97158012682676E-01-4.41709746642829E-01*i)*zz1
+ (-2.11981526746876E-01 + 9.77273673193985E-01*i)*zz2
+ (7.37616926767541E-01 + 6.75219423110746E-01*i)*zz3 + 1;
 + (-3.53404925407865E-01-1.02449352102111E-02*i)*x1
+ (-3.23378971672498E-01-1.42919700111768E-01*i)*x2
+ (2.20216662912369E-01-2.76594687902244E-01*i)*x3
+ (8.35397253138817E-02-3.43542012415485E-01*i)*x4
+ (-2.67791727313329E-01-2.30841050904175E-01*i)*zz1
```

(continues on next page)

(continued from previous page)

```
+ (3.28014183926858E-01-1.31934435015266E-01*i)*zz2
+ (-1.52787277524516E-01 + 3.18835455723868E-01*i)*zz3 - 3.53553390593274E-01;
 + (-4.95361781887585E-01 + 5.82835865995319E-03*i)*x1
+ (1.04868230499720E-01 + 2.82848378492330E-01*i)*x2
+ (-1.92957771968200E-01 + 1.72592823677708E-01*i)*x3
+ (1.50836540788987E-01-4.65440876777183E-01*i)*x4
+ (2.59311098918677E-01-3.82322947183237E-02*i)*zz1
+ (-2.61338053548032E-01-1.91269889240060E-01*i)*zz2
+ (-3.36536283081467E-01-7.29526150129006E-02*i)*zz3
+ (2.54532009330191E-01 + 1.49441343387202E-02*i);
 + (1.15184561011707E-01 + 1.13377932381136E-01*i)*x1
+ (-5.17726679477236E-01-8.32947479684604E-02*i)*x2
+ (-3.72758479505006E-01-2.38502017467544E-02*i)*x3
+ (-9.43470497132125E-02-1.99900948005935E-01*i)*x4
+ (-2.38409509176001E-01-2.75632135792588E-01*i)*zz1
+ (-1.20802218865972E-01 + 2.04083624753821E-01*i)*zz2
+ (1.70161843122080E-02-4.70880850401632E-01*i)*zz3
+ (8.75322932557317E-02-3.02958515664302E-01*i);
  the generic points :
  Solution 1 :
  t :  1.00000000000000E+00   0.00000000000000E+00
  m : 1
  the solution for t :
   x1 :  1.00000000000000E+00   8.42939386835822E-17
   x2 : -4.63024527132596E-01  -8.07963582510971E-01
   x3 :  1.38259558372538E+00   3.26937976582811E-01
   x4 :  2.04312757081212E-01   7.58951450601146E-01
   zz1 : -7.86154493035376E-16   7.63862347412176E-16
   zz2 :  2.83697554118352E-16   9.21340974182504E-16
   zz3 :  1.05678537850043E-16   6.86142378280738E-17
  == err :  6.527E-15 = rco :  1.666E-02 = res :  3.737E-15 =
```

3.14.6 diagonal homotopies

An alternative to solving polynomial systems from the top to the bottom, is to start intersection the equations one after the other. Consider the intersection of a sphere with a cylinder.

```
sphere = 'X^2 + Y^2 + Z^2 - 1;'
cylinder = 'X^2 + 1.0e-14*Y^2 + (Z - 0.5)^2 - 1;'
```

Observe the tiny coefficient of `Y^2` which is a trick to align the symbols of the two equations. The upper case letters of the variables are needed for the `verify` not to be confused by `zz1`.

First, witness sets are constructed for the two hypersurfaces.

```
from phcpy.sets import double_hypersurface_set
```

In each instance we check the number of generic points computed, which should be 2 in both.

```
(spheqs, sphpts) = double_hypersurface_set(3, sphere)
len(sphpts)
```

which indeed shows 2 and then also

```
(cyleqs, cylpts) = double_hypersurface_set(3, cylinder)
len(cylpts)
```

shows 2 as the number of generic points.

```
from phcpy.diagonal import double_diagonal_solve
quaeqs, quapts = double_diagonal_solve(3, 2, spheqs, sphpts, 2, cyleqs, cylpts)
```

The polynomials in the computed witness set of the intersection are ...”

```
for pol in quaeqs:
    print(pol)
```

shown below:

```
 + X^2 + Y^2 + Z^2 + (1.66568720348346E-01 + 9.86029848129109E-01*i)*zz1 - 1;
 + X^2 + 1.00000000000000E-14*Y^2 + Z^2 - Z + (7.86376622880735E-01 + 6.17747365018006E-
↪01*i)*zz1 - 7.50000000000000E-01;
zz1;
 + (1.58876905105528E-01-7.26285688514595E-01*i)*X + (-8.16467339319674E-01 + 1.
↪71502319167546E+00*i)*Y + (-4.76416867298768E-02 + 4.42262587408809E-01*i)*Z + (5.
↪34984994186327E-01 + 8.44861560254374E-01*i)*zz1+(-8.46146634046559E-01 + 5.
↪32950160607611E-01*i);
```

and the generic points in the computed witness set are printed with

```
for (idx, sol) in enumerate(quapts):
    print('Solution', idx+1, ':')
    print(sol)
```

and the output is

```
Solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 X :  9.91243806839434E-01  -1.45160013935997E-02
 Y : -1.36376604565112E-01  -3.29060036857026E-01
 Z :  3.39681929583638E-01  -8.97521810492629E-02
 zz1 :  0.00000000000000E+00   0.00000000000000E+00
== err :  3.374E-16 = rco :  4.134E-02 = res :  1.769E-16 =
Solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 X : -7.67613124615721E-01   1.54768674480021E-01
 Y : -6.69973298698680E-01  -1.30010400626017E-01
 Z : -1.81961516698249E-01  -1.74206993945098E-01
 zz1 :  0.00000000000000E+00   0.00000000000000E+00
== err :  3.448E-16 = rco :  5.434E-02 = res :  3.886E-16 =
Solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
```

(continues on next page)

(continued from previous page)

```
 X :  4.60320208343188E+00   4.38623359269609E+00
 Y :  9.59637373859308E-01   2.36891289291490E+00
 Z :  4.94084440491082E+00  -4.54659469491659E+00
 zz1 :  0.00000000000000E+00   0.00000000000000E+00
== err :  3.427E-14 = rco :  6.196E-04 = res :  2.465E-14 =
Solution 4 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 X :  6.32992398543307E+00   2.24044198839475E+00
 Y :  2.07264148009942E+00  -1.51003268649074E+00
 Z : -1.76564399075824E+00   6.25951276465330E+00
 zz1 :  0.00000000000000E+00   0.00000000000000E+00
== err :  4.575E-14 = rco :  1.103E-03 = res :  2.687E-14 =
```

Four generic points are obtained in the computed witness set, as the intersection of a sphere with a cylinder is a quartic.

3.15 Code Snippets

The functions are based on the code snippets of the menus of the Jupyter notebook extension of PHCpy. Every function is self contained and illustrates one particular feature.

3.15.1 the blackbox solver

```
def solve_random_trinomials():
    """
    Illustrates the solution of random trinomials.
    """
    from phcpy.solver import random_trinomials
    f = random_trinomials()
    for pol in f: print(pol)
    from phcpy.solver import solve
    sols = solve(f)
    for sol in sols: print(sol)
    print(len(sols), "solutions found")
```

```
def solve_specific_trinomials():
    """
    Illustrates the solution of specific trinomials.
    """
    f = ['x^2*y^2 + 2*x - 1;', 'x^2*y^2 - 3*y + 1;']
    from phcpy.solver import solve
    sols = solve(f)
    for sol in sols: print(sol)
```

```
def solution_from_string_to_dictionary():
    """
    Illustrates the dictionary format of a solution.
```

(continues on next page)

(continued from previous page)

```
    """
    p = ['x + y - 1;', '2*x - 3*y + 1;']
    from phcpy.solver import solve
    sols = solve(p)
    print(sols[0])
    from phcpy.solutions import strsol2dict
    dsol = strsol2dict(sols[0])
    print(dsol.keys())
    for key in dsol.keys(): print('the value for', key, 'is', dsol[key])
```

```
def verify_by_evaluation():
    """
    Illustrates the verification of solutions by evaluation.
    """
    p = ['x + y - 1;', '2*x - 3*y + 1;']
    from phcpy.solver import solve
    sols = solve(p)
    from phcpy.solutions import strsol2dict, evaluate
    dsol = strsol2dict(sols[0])
    eva = evaluate(p, dsol)
    for val in eva: print(val)
```

```
def make_a_solution():
    """
    Illustrates the making of a solution.
    """
    from phcpy.solutions import make_solution
    s0 = make_solution(['x', 'y'], [float(3.14), complex(0, 2.7)])
    print(s0)
    s1 = make_solution(['x', 'y'], [int(2), int(3)])
    print(s1)
```

```
def filter_solution_lists():
    """
    Illustrates the filtering of solution lists.
    """
    from phcpy.solutions import make_solution, is_real, filter_real
    s0 = make_solution(['x', 'y'], [float(3.14), complex(0, 2.7)])
    print(is_real(s0, 1.0e-8))
    s1 = make_solution(['x', 'y'], [int(2), int(3)])
    print(is_real(s1, 1.0e-8))
    realsols = filter_real([s0, s1], 1.0e-8, 'select')
    for sol in realsols: print(sol)
```

```
def coordinates_names_values():
    """
    Illustrates coordinates, names, values of a solution.
    """
    from phcpy.solver import solve
    p = ['x^2*y^2 + x + 1;', 'x^2*y^2 + y + 1;']
    s = solve(p)
```

(continues on next page)

(continued from previous page)

```
    print(s[0])
    from phcpy.solutions import coordinates, make_solution
    (names, values) = coordinates(s[0])
    print(names)
    print(values)
    s0 = make_solution(names, values)
    print(s0)
```

```
def fix_retrieve_seed():
    """
    Illustrates fixing and retrieving the seed.
    """
    from phcpy.dimension import set_seed, get_seed
    set_seed(2024)
    print(get_seed())
```

```
def solve_with_four_threads():
    """
    Ilustrates solving with four threads.
    """
    from phcpy.solver import solve
    from phcpy.families import cyclic
    nbrt = 4 # number of tasks
    pols = cyclic(6)
    print('solving the cyclic 6-roots problem :')
    for pol in pols: print(pol)
    from datetime import datetime
    starttime = datetime.now()
    sols = solve(pols)
    stoptime = datetime.now()
    elapsed = stoptime - starttime
    print(f'elapsed time with no multithreading :')
    print(elapsed)
    starttime = datetime.now()
    sols = solve(pols, tasks=nbrt)
    stoptime = datetime.now()
    elapsed = stoptime - starttime
    print(f'elapsed time with {nbrt} threads :')
    print(elapsed)
```

```
def four_root_counts():
    """
    Illustrates four root counts.
    """
    f = ['x^3*y^2 + x*y^2 + x^2;', 'x^5 + x^2*y^3 + y^2;']
    from phcpy.starters import total_degree
    print('the total degree :', total_degree(f))
    from phcpy.starters import m_homogeneous_bezout_number as mbz
    (bz, part) = mbz(f)
    print('a multihomogeneous Bezout number :', bz)
    from phcpy.starters import linear_product_root_count as lrc
```

(continues on next page)

(continued from previous page)

```
    (rc, ssrc) = lrc(f)
    print('a linear-product root count :', rc)
    from phcpy.volumes import stable_mixed_volume
    (mv, smv) = stable_mixed_volume(f)
    print('the mixed volume :', mv)
    print('the stable mixed volume :', smv)
```

```
def newton_and_deflation():
    """
    Illustrates Newton's method and deflation.
    """
    p = ['(29/16)*x^3 - 2*x*y;', 'x^2 - y;']
    from phcpy.solutions import make_solution
    s = make_solution(['x', 'y'],[float(1.0e-6), float(1.0e-6)])
    print(s)
    from phcpy.deflation import double_newton_step
    s2 = double_newton_step(p, [s])
    print(s2[0])
    s3 = double_newton_step(p, s2)
    print(s3[0])
    from phcpy.deflation import double_deflate
    sd = double_deflate(p, [s])
    print(sd[0])
```

```
def overconstrained_deflation():
    """
    Illustrates deflation of an overconstrained system.
    """
    from phcpy.solutions import make_solution
    from phcpy.deflation import double_deflate
    sol = make_solution(['x', 'y'], [float(1.0e-6), float(1.0e-6)])
    print(sol)
    pols = ['x**2;', 'x*y;', 'y**2;']
    sols = double_deflate(pols, [sol], tolrnk=1.0e-8)
    print(sols[0])
    sols = double_deflate(pols, [sol], tolrnk=1.0e-4)
    print(sols[0])
```

```
def equation_and_variable_scaling():
    """
    Illustrates equation and variable scaling.
    """
    print('solving without scaling ...')
    from phcpy.solver import solve
    p = ['0.000001*x^2 + 0.000004*y^2 - 4;', '0.000002*y^2 - 0.001*x;']
    psols = solve(p)
    for sol in psols: print(sol)
    print('solving after scaling ...')
    from phcpy.scaling import double_scale_system as scalesys
    from phcpy.scaling import double_scale_solutions as scalesols
    (q, c) = scalesys(p)
```

(continues on next page)

(continued from previous page)

```
    print('the scaled polynomial system :')
    for pol in q: print(pol)
    qsols = solve(q)
    ssols = scalesols(len(q), qsols, c)
    for sol in ssols: print(sol)
```

3.15.2 path trackers

```
def total_degree_start_system():
    """
    The product of the degrees of the polynomials (the total degree)
    provide an upper bound on the number of solutions.
    A total degree start system is a simple system that has
    as many solutions as the product of the degrees.
    """
    from phcpy.starters import total_degree
    from phcpy.starters import total_degree_start_system
    from phcpy.trackers import double_track
    p = ['x^2 + 4*y^2 - 4;', '2*y^2 - x;']
    d = total_degree(p)
    print('the total degree :', d)
    (q, qsols) = total_degree_start_system(p)
    print('the number of start solutions :', len(qsols))
    print('the start system :', q)
    s = double_track(p, q, qsols)
    print('the number of solutions :', len(s))
    for sol in s: print(sol)
```

```
def track_one_path():
    """
    Illustrates the tracking of one solution path.
    The order in which the solutions appear at the end
    depends on the gamma constant.
    """
    from phcpy.starters import total_degree_start_system
    from phcpy.trackers import double_track
    p = ['x^2 + 4*y^2 - 4;', '2*y^2 - x;']
    (q, qsols) = total_degree_start_system(p)
    g1, s1 = double_track(p, q, [qsols[2]])
    print('first gamma :', g1)
    print(s1[0])
    g2, s2 = double_track(p, q, [qsols[2]])
    print('second gamma :', g2)
    print(s2[0])
```

```
def track_with_fixed_gamma():
    """
    Fixing the gamma in the homotopies fixes the order
    in which the solutions appear at the end of the paths.
    """
```

(continues on next page)

(continued from previous page)

```
    from phcpy.starters import total_degree_start_system
    from phcpy.trackers import double_track
    p = ['x^2 + 4*y^2 - 4;', '2*y^2 - x;']
    (q, qsols) = total_degree_start_system(p)
    g3, s3 = double_track(p, q, [qsols[2]], \
        gamma=complex(0.824372806319,0.56604723848934))
    print('gamma :', g3)
    print('the solution at the end:')
    print(s3[0])
```

```
def get_next_point_on_path():
    """
    A step-by-step path tracker gives control to the user
    who can ask for the next point on a path.
    """
    from phcpy.starters import total_degree_start_system
    p = ['x**2 + 4*x**2 - 4;', '2*y**2 - x;']
    (q, s) = total_degree_start_system(p)
    from phcpy.trackers import initialize_double_tracker
    from phcpy.trackers import initialize_double_solution
    from phcpy.trackers import next_double_solution
    initialize_double_tracker(p, q)
    initialize_double_solution(len(p), s[0])
    s1 = next_double_solution()
    print('the next point on the solution path :')
    print(s1)
    print(next_double_solution())
    print(next_double_solution())
    initialize_double_solution(len(p), s[1])
    points = [next_double_solution() for i in range(11)]
    from phcpy.solutions import strsol2dict
    dicpts = [strsol2dict(sol) for sol in points]
    xvals = [sol['x'] for sol in dicpts]
    print('the x-coordinates on the path :')
    for x in xvals: print(x)
```

```
def plot_trajectories():
    """
    The step-by-step path tracker is applied to plot the
    trajectories of the solutions using matplotlib.
    """
    import matplotlib.pyplot as plt
    p = ['x^2 + y - 3;', 'x + 0.125*y^2 - 1.5;']
    print('constructing a total degree start system ...')
    from phcpy.starters import total_degree_start_system
    q, qsols = total_degree_start_system(p)
    print('number of start solutions :', len(qsols))
    from phcpy.trackers import initialize_double_tracker
    from phcpy.trackers import initialize_double_solution
    from phcpy.trackers import next_double_solution
    initialize_double_tracker(p, q, False)
```

(continues on next page)

(continued from previous page)

```
    from phcpy.solutions import strsol2dict
    plt.ion()
    fig = plt.figure()
    for k in range(len(qsols)):
        if(k == 0):
           axs = fig.add_subplot(221)
        elif(k == 1):
           axs = fig.add_subplot(222)
        elif(k == 2):
            axs = fig.add_subplot(223)
        elif(k == 3):
           axs = fig.add_subplot(224)
        startsol = qsols[k]
        initialize_double_solution(len(p),startsol)
        dictsol = strsol2dict(startsol)
        xpoints =  [dictsol['x']]
        ypoints =  [dictsol['y']]
        for k in range(300):
            ns = next_double_solution()
            dictsol = strsol2dict(ns)
            xpoints.append(dictsol['x'])
            ypoints.append(dictsol['y'])
            tval = dictsol['t'].real
            if(tval >= 1.0):
                break
        print(ns)
        xre = [point.real for point in xpoints]
        yre = [point.real for point in ypoints]
        axs.set_xlim(min(xre)-0.3, max(xre)+0.3)
        axs.set_ylim(min(yre)-0.3, max(yre)+0.3)
        dots, = axs.plot(xre,yre,'r-')
        fig.canvas.draw()
    fig.canvas.draw()
    ans = input('hit return to continue')
```

```
def polyhedral_homotopies():
    """
    Polyhedral homotopies solve random coefficient systems
    tracking an optimal number of solution paths,
    that is equal to the mixed volume.
    """
    from phcpy.volumes import mixed_volume
    from phcpy.volumes import double_polyhedral_homotopies
    from phcpy.trackers import double_track
    p = ['x^3*y^2 - 3*x^3 + 7;','x*y^3 + 6*y^3 - 9;']
    print('the mixed volume :', mixed_volume(p))
    (q, qsols) = double_polyhedral_homotopies()
    print('the number of start solutions :', len(qsols))
    gamma, psols = double_track(p, q, qsols)
    print('the number of solutions at the end :', len(psols))
    for sol in psols: print(sol)
```

3.15.3 sweep homotopies

```
def quadratic_turning_point():
    """
    Arc length parameter continuation is applied in a real sweep
    which ends at a quadratic turning point.
    """
    from phcpy.sweepers import double_real_sweep
    from phcpy.solutions import make_solution
    circle = ['x^2 + y^2 - 1;', 'y*(1-s) + (y-2)*s;']
    first = make_solution(['x', 'y', 's'], [1.0, 0.0, 0.0])
    second = make_solution(['x', 'y', 's'], [-1.0, 0.0, 0.0])
    startsols = [first, second]
    newsols = double_real_sweep(circle, startsols)
    for sol in newsols: print(sol)
```

```
def complex_parameter_homotopy_continuation():
    """
    Sweeps the parameter space with a convex linear combination
    of the parameters.  By a random gamma constant, no singularities
    are encountered during this complex sweep.
    """
    from phcpy.sweepers import double_complex_sweep
    from phcpy.solutions import make_solution
    circle = ['x^2 + y^2 - 1;']
    first = make_solution(['x', 'y'], [1.0, 0.0])
    second = make_solution(['x', 'y'], [-1.0, 0.0])
    startsols = [first, second]
    par = ['y']
    start = [0, 0]
    target = [2, 0]
    newsols = double_complex_sweep(circle, startsols, 2, par, start, target)
    for sol in newsols: print(sol)
```

3.15.4 Schubert calculus

```
def lines_meeting_four_lines():
    """
    Applies Pieri homotopies to compute all lines meeting
    four given lines in 3-space.
    """
    from phcpy.schubert import pieri_root_count
    from phcpy.schubert import random_complex_matrix, run_pieri_homotopies
    (m, p, q) = (2, 2, 0)
    dim = m*p + q*(m+p)
    roco = pieri_root_count(m, p, q)
    print('the root count :', roco)
    L = [random_complex_matrix(m+p, m) for _ in range(dim)]
    (f, fsols) = run_pieri_homotopies(m, p, q, L)
    for sol in fsols: print(sol)
    print('number of solutions :', len(fsols))
```

```
def line_producing_interpolating_curves():
    """
    Applies Pieri homotopies to compute line producing curves,
    interpolating at given lines in 3-space.
    """
    from phcpy.schubert import pieri_root_count
    from phcpy.schubert import random_complex_matrix, run_pieri_homotopies
    (m, p, q) = (2, 2, 1)
    dim = m*p + q*(m+p)
    roco = pieri_root_count(m, p, q)
    print('the root count :', roco)
    L = [random_complex_matrix(m+p, m) for _ in range(dim)]
    points = random_complex_matrix(dim, 1)
    (f, fsols) = run_pieri_homotopies(m, p, q, L, 0, points)
    print('number of solutions :', len(fsols))
```

```
def resolve_some_schubert_conditions():
    """
    Resolves an example of a Schubert condition,
    applying the Littlewood-Richardson rule.
    """
    from phcpy.schubert import resolve_schubert_conditions
    brackets = [[2, 4, 6], [2, 4, 6], [2, 4, 6]]
    roco = resolve_schubert_conditions(6, 3, brackets)
    print('number of solutions :', roco)
```

```
def solve_generic_schubert_problem():
    """
    Runs the Littlewood-Richardson homotopies to solve
    a generic instance of a Schubert problem.
    """
    brackets = [[2, 4, 6], [2, 4, 6], [2, 4, 6]]
    from phcpy.schubert import double_littlewood_richardson_homotopies as lrh
    (count, flags, sys, sols) = lrh(6, 3, brackets, verbose=False)
    print('the root count :', count)
    for sol in sols: print(sol)
    print('the number of solutions :', len(sols))
```

3.15.5 power series expansions

```
def example4pade():
    """
    The function f(z) = ((1 + 1/2*z)/(1 + 2*z))^(1/2) is
    a solution x(s) of (1-s)*(x^2 - 1) + s*(3*x^2 - 3/2) = 0
    """
    from phcpy.solutions import make_solution
    from phcpy.series import double_newton_at_point
    from phcpy.series import double_pade_approximants
    pol = ['(x^2 - 1)*(1-s) + (3*x^2 - 3/2)*s;']
    variables = ['x', 's']
```

(continues on next page)

(continued from previous page)

```
    sol1 = make_solution(variables, [1, 0])
    sol2 = make_solution(variables, [-1, 0])
    sols = [sol1, sol2]
    print('start solutions :')
    for sol in sols: print(sol)
    srs = double_newton_at_point(pol, sols, idx=2)
    print('The series :')
    for ser in srs: print(ser)
    pad = double_pade_approximants(pol, sols, idx=2)
    print('the Pade approximants :')
    for app in pad: print(app)
```

```
def viviani_expansion(vrblvl=0):
    """
    Computes the power series expansion for the Viviani curve,
    from a natural parameter perspective, in double precision.
    """
    from phcpy.series import double_newton_at_series
    pols = [ '2*t^2 - x;', \
             'x^2 + y^2 + z^2 - 4;' , \
             '(x-1)^2 + y^2 - 1;']
    lser = [ '2*t^2;', '2*t;', '2;']
    nser = double_newton_at_series(pols, lser, maxdeg=12, nbr=8)
    variables = ['x', 'y', 'z']
    for (var, pol) in zip(variables, nser): print(var, '=', pol)
```

```
def apollonius_expansions():
    """
    Compare the series expansions at two solutions
    for the problem of Apollonius.
    """
    from phcpy.series import double_newton_at_series
    pols = [ 'x1^2 + 3*x2^2 - r^2 - 2*r - 1;', \
             'x1^2 + 3*x2^2 - r^2 - 4*x1 - 2*r + 3;', \
    '3*t^2 + x1^2 - 6*t*x2 + 3*x2^2 - r^2 + 6*t - 2*x1 - 6*x2 + 2*r + 3;']
    lser1 = ['1;', '1 + 0.536*t;', '1 + 0.904*t;']
    lser2 = ['1;', '1 + 7.464*t;', '1 + 11.196*t;']
    nser1 = double_newton_at_series(pols, lser1, idx=4, nbr=7)
    nser2 = double_newton_at_series(pols, lser2, idx=4, nbr=7)
    variables = ['x', 'y', 'z']
    print('the first solution series :')
    for (var, pol) in zip(variables, nser1): print(var, '=', pol)
    print('the second solution series :')
    for (var, pol) in zip(variables, nser2): print(var, '=', pol)
```

3.15.6 positive dimensional solution sets

```
def twisted_cubic_witness_set():
    """
    Makes a witness set for the twisted cubic.
    """
    twisted = ['x^2 - y;', 'x^3 - z;']
    from phcpy.sets import double_embed
    embtwist = double_embed(3, 1, twisted)
    print('embedded system augmented with a random hyperplane :')
    for pol in embtwist:
        print(pol)
    from phcpy.solver import solve
    sols = solve(embtwist)
    print('number of generic points :', len(sols))
    for sol in sols: print(sol)
```

```
def homotopy_membership_test():
    """
    Homotopies to decide whether a point belongs to a positive
    dimensional solution set.
    """
    from phcpy.families import cyclic
    c4 = cyclic(4)
    from phcpy.sets import double_embed
    c4e1 = double_embed(4, 1, c4)
    from phcpy.solver import solve
    sols = solve(c4e1)
    from phcpy.solutions import filter_zero_coordinates as filter
    genpts = filter(sols, 'zz1', 1.0e-8, 'select')
    print('generic points on the cyclic 4-roots set :')
    for sol in genpts: print(sol)
    from phcpy.sets import double_membertest
    pt0 = [1, 0, -1, 0, 1, 0, -1, 0]
    ismbr = double_membertest(c4e1, sols, 1, pt0)
    print('Is', pt0, 'a member?', ismbr)
    pt1 = [1, 0, 1, 0, -1, 0, -1, 0]
    ismbr = double_membertest(c4e1, sols, 1, pt1)
    print('Is', pt1, 'a member?', ismbr)
```

```
def factor_a_cubic():
    """
    Illustrates the numerical factorization of a cubic.
    """
    cubic = '(x+1)*(x^2 + y^2 + 1);'
    from phcpy.sets import double_hypersurface_set
    (wit, pts) = double_hypersurface_set(2, cubic)
    for pol in wit:
        print(pol)
    print('number of witness points :', len(pts))
    for (idx, sol) in enumerate(pts):
        print('Solution', idx+1, ':')
```

(continues on next page)

(continued from previous page)

```
        print(sol)
    from phcpy.factor import double_monodromy_breakup, write_factorization
    deco = double_monodromy_breakup(wit, pts, dim=1)
    write_factorization(deco)
```

```
def cascades_of_homotopies():
    """
    A cascade of homotopies computes generic points on all positive
    components of the solution set, for all dimensions.
    """
    pol1 = '(x^2 + y^2 + z^2 - 1)*(y - x^2)*(x - 0.5);'
    pol2 = '(x^2 + y^2 + z^2 - 1)*(z - x^3)*(y - 0.5);'
    pol3 = '(x^2 + y^2 + z^2 - 1)*(z - x*y)*(z - 0.5);'
    pols = [pol1, pol2, pol3]
    from phcpy.cascades import double_top_cascade, double_cascade_filter
    (embpols, sols0, sols1) = double_top_cascade(3, 2, pols)
    print('at dimension 2, degree :', len(sols0))
    (wp1, ws0, ws1) = double_cascade_filter(2, embpols, sols1, tol=1.0e-8)
    print('at dimension 1, candidate generic points :', len(ws0))
    (wp0, ws0, ws1) = double_cascade_filter(1, wp1, ws1, tol=1.0e-8)
    print('candidate isolated points :', len(ws0))
```

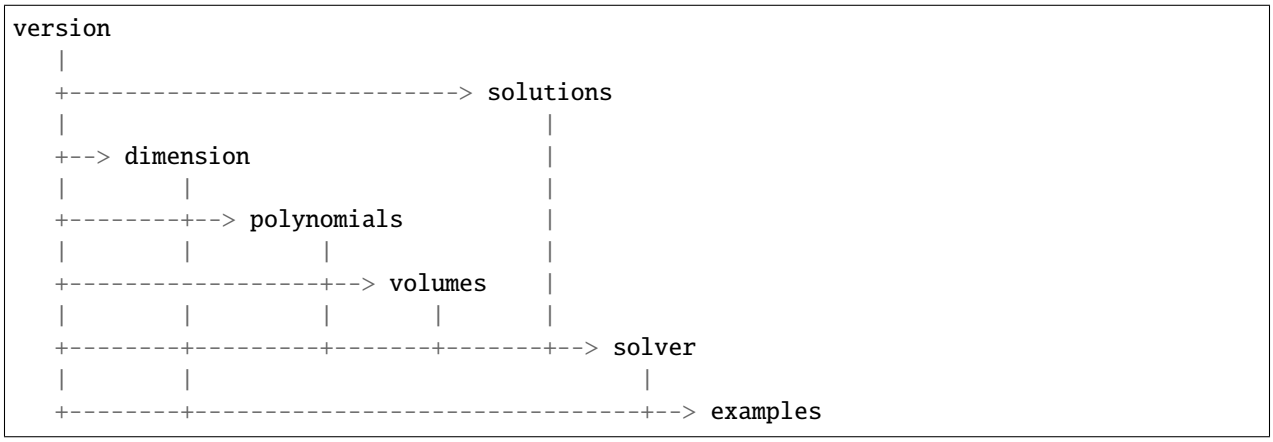
```
def numerical_irreducible_decomposition():
    """
    Runs a test example on a numerical irreducible decomposition.
    """
    pol0 = '(x1-1)*(x1-2)*(x1-3)*(x1-4);'
    pol1 = '(x1-1)*(x2-1)*(x2-2)*(x2-3);'
    pol2 = '(x1-1)*(x1-2)*(x3-1)*(x3-2);'
    pol3 = '(x1-1)*(x2-1)*(x3-1)*(x4-1);'
    pols = [pol0, pol1, pol2, pol3]
    from phcpy.decomposition import solve, write_decomposition
    deco = solve(pols, verbose=False)
    write_decomposition(deco)
```

```
def sphere_cylinder_intersection():
    """
    A sphere intersected by a cylinder defines a quartic curve.
    """
    sphere = 'X^2 + Y^2 + Z^2 - 1;'
    cylinder = 'X^2 + 1.0e-14*Y^2 + (Z - 0.5)^2 - 1;'
    from phcpy.sets import double_hypersurface_set
    (spheqs, sphpts) = double_hypersurface_set(3, sphere)
    (cyleqs, cylpts) = double_hypersurface_set(3, cylinder)
    from phcpy.diagonal import double_diagonal_solve
    quaeqs, quapts = double_diagonal_solve\
        (3, 2, spheqs, sphpts, 2, cyleqs, cylpts)
    for pol in quaeqs: print(pol)
    for sol in quapts: print(sol)
```

CHAPTER 4

Programmer Manual

The code is spread over more than twenty modules. The directed acyclic graph shows the dependencies of the first seven modules:



At the root is the version module. If version.py works, then the interfacing with libPHCpack works. Four other modules are needed for the blackbox solver.

In addition to functions, each module has test functions, starting with `test_`. Each test function returns the number of failed tests. The `main()` of each module runs all tests.

4.1 the version module

The PHCpack version string is retrieved via this module, along with some utility functions for the basic type conversions.

Tests the retrieval of the version string of the PHCpack library, using ctypes, in a platform independent manner. The function getPHCmod() returns the library module for the three supported platforms. The functions int4a2str() and str2int4a() convert integer arrays to strings and string to integer arrays, using the ctypes string buffer type. The function int4a2nbr() converts a list of integers to the string buffer representation of the 32-bit integer array version.

version.**get_phcfun**(*vrblvl=0*)

Returns phcpy.phc, or if that not works, returns get_phcfun_fromlib()

version.**get_phcfun_fromlib**(*vrblvl=0*)

Returns the proper function according to the platform. For the correct execution, the file libPHCpack, with the extension .so, .dylib, or .dll must be present.

version.**int4a2nbr**(*data*, *vrblvl=0*)

Given in data is a Python list of integers, returns the encoding in a ctypes string buffer, for conversion into an 32-bit integer array. If vrblvl > 0, then results of intermediate steps are shown, otherwise the function remains silent.

version.**int4a2str**(*data*, *vrblvl=0*)

Given in data is an integer array stored as a ctypes string buffer, returns the string which represent the data. The '4a' refers to the 4-byte (or 32-bit) integer array. If vrblvl > 0, then results of intermediate steps are printed, otherwise, the function remains silent.

version.**main**()

Prints the version string and runs two tests.

version.**nbr2int4a**(*data*, *vrblvl=0*)

Given in data is a 32-bit integer array, in a ctypes string buffer. Returns the list of python integers.

version.**str2int4a**(*data*, *vrblvl=0*)

Given in data is a string, stored as a Python string, returns the 32-bit integer array representation, stored as a ctypes string buffer. If vrblvl > 0, then results of intermediate steps are printed, otherwise, the function remains silent.

version.**test_byte_strings**(*vrblvl=0*)

Tests the conversion of a string into a string buffer and then backwards.

version.**test_integer_encodings**(*vrblvl=0*)

Tests the encoding of a list of integers as a ctypes string buffer.

version.**version_string**(*vrblvl=0*)

Returns the version string of PHCpack. If vrblvl > 0, then the conversions between strings and integer arrays are verified via the ctypes string buffer types.

4.2 a blackbox solver for isolated solutions

The three most essential modules to solve polynomial systems are the `solver` module, which exports the blackbox solver, the `polynomials` module, to define the input, and the `solutions` module, to parse the computed solutions.

4.2.1 functions in the module dimension

In addition to the number of polynomials in the system, the dimension module controls some other important numbers for each run: the seed of the random number generators and the number of available cores.

Exports the dimension of the system of polynomials. Setting the dimension allocates memory to store polynomials.

`dimension.`**`get_core_count`**`(`*`vrblvl=0`*`)`

Returns the number of available cores.

`dimension.`**`get_double_dimension`**`(`*`vrblvl=0`*`)`

Returns the number of polynomials in double precision.

`dimension.`**`get_double_double_dimension`**`(`*`vrblvl=0`*`)`

Returns the number of polynomials in double double precision.

`dimension.`**`get_double_double_laurent_dimension`**`(`*`vrblvl=0`*`)`

Returns the number of Laurent polynomials in double double precision.

`dimension.`**`get_double_laurent_dimension`**`(`*`vrblvl=0`*`)`

Returns the number of Laurent polynomials in double precision.

`dimension.`**`get_quad_double_dimension`**`(`*`vrblvl=0`*`)`

Returns the number of polynomials in quad double precision.

`dimension.`**`get_quad_double_laurent_dimension`**`(`*`vrblvl=0`*`)`

Returns the number of Laurent polynomials in quad double precision.

`dimension.`**`get_seed`**`(`*`vrblvl=0`*`)`

Returns the seed used to generate random numbers.

`dimension.`**`main`**`()`

Runs tests on set/get dimension and set/get seed.

`dimension.`**`set_double_dimension`**`(`*`dim`*`, `*`vrblvl=0`*`)`

Sets the number of polynomials in double precision to the value of the first parameter dim.

`dimension.`**`set_double_double_dimension`**`(`*`dim`*`, `*`vrblvl=0`*`)`

Sets the number of polynomials in double double precision to the value of the first parameter dim.

`dimension.`**`set_double_double_laurent_dimension`**`(`*`dim`*`, `*`vrblvl=0`*`)`

Sets the number of Laurent polynomials in double double precision to the value of the first parameter dim.

`dimension.`**`set_double_laurent_dimension`**`(`*`dim`*`, `*`vrblvl=0`*`)`

Sets the number of Laurent polynomials in double precision to the value of the first parameter dim.

`dimension.`**`set_quad_double_dimension`**`(`*`dim`*`, `*`vrblvl=0`*`)`

Sets the number of polynomials in quad double precision to the value of the first parameter dim.

`dimension.`**`set_quad_double_laurent_dimension`**`(`*`dim`*`, `*`vrblvl=0`*`)`

Sets the number of Laurent polynomials in quad double precision to the value of the first parameter dim.

dimension.**set_seed**(*seed*, *vrblvl=0*)

Sets the seed for the random number generators to seed.

dimension.**test_core_count**(*vrblvl=0*)

Tests if the number of available cores is positive. The verbose level is defined by vrblvl.

dimension.**test_dimension**(*vrblvl=0*)

Tests setting and getting the dimension. The verbose level is defined by vrblvl.

dimension.**test_double_dimension**(*vrblvl=0*)

Tests setting and getting the dimension for systems in double precision. The verbose level is defined by vrblvl.

dimension.**test_double_double_dimension**(*vrblvl=0*)

Tests setting and getting the dimension for systems in double double precision. The verbose level is defined by vrblvl.

dimension.**test_double_double_laurent_dimension**(*vrblvl=0*)

Tests setting and getting the dimension for Laurent systems in double double precision. The verbose level is defined by vrblvl.

dimension.**test_double_laurent_dimension**(*vrblvl=0*)

Tests setting and getting the dimension for Laurent systems in double precision. The verbose level is defined by vrblvl.

dimension.**test_quad_double_dimension**(*vrblvl=0*)

Tests setting and getting the dimension for systems in quad double precision. The verbose level is defined by vrblvl.

dimension.**test_quad_double_laurent_dimension**(*vrblvl=0*)

Tests setting and getting the dimension for Laurent systems in quad double precision. The verbose level is defined by vrblvl.

dimension.**test_seed**(*vrblvl=0*)

Tests the setting and getting of the seed. The verbose level is defined by vrblvl.

4.2.2 functions in the module polynomials

The module polynomials exports the definition of a class to represent systems of polynomials.

Exports the definition of polynomial systems, in double, double double, and quad double precision. Also polynomials with negative exponents, the so-called Laurent polynomials, are supported.

polynomials.**check_semicolons**(*pols*, *vrblvl=0*)

Counts the number of semicolons and writes a warning if the number of semicolons in pols does not match len(pols). This warning may prevent unwanted concatenation, e.g., as in pols = ['x^2 + 4*y^2 - 4;' '2*y^2 - x;'] where the omitted comma results in one polynomial in two variables.

polynomials.**clear_double_double_laurent_system**(*vrblvl=0*)

Clears the Laurent system set in double double precision.

polynomials.**clear_double_double_syspool**(*vrblvl=0*)

Clears the systems pool in double double precision.

polynomials.**clear_double_double_system**(*vrblvl=0*)

Clears the system set in double double precision.

polynomials.**clear_double_laurent_system**(*vrblvl=0*)

Clears the Laurent system set in double precision.

polynomials.**clear_double_syspool**(*vrblvl=0*)

Clears the systems pool in double precision.

polynomials.**clear_double_system**(*vrblvl=0*)

Clears the system set in double precision.

polynomials.**clear_quad_double_laurent_system**(*vrblvl=0*)

Clears the Laurent system set in quad double precision.

polynomials.**clear_quad_double_syspool**(*vrblvl=0*)

Clears the systems pool in quad double precision.

polynomials.**clear_quad_double_system**(*vrblvl=0*)

Clears the system set in quad double precision.

polynomials.**clear_symbol_table**(*vrblvl=0*)

Clears the table of symbols used to represent polynomials.

polynomials.**copy_from_double_double_syspool**(*idx*, *vrblvl=0*)

Copies the system in double double precision at position idx in the systems pool to the defined system in double double precision.

polynomials.**copy_from_double_syspool**(*idx*, *vrblvl=0*)

Copies the system in double precision at position idx in the systems pool to the defined system in double precision.

polynomials.**copy_from_quad_double_syspool**(*idx*, *vrblvl=0*)

Copies the system in quad double precision at position idx in the systems pool to the defined system in quad double precision.

polynomials.**copy_to_double_double_syspool**(*idx*, *vrblvl=0*)

Copies the system set in double double precision to position idx in the systems pool.

polynomials.**copy_to_double_syspool**(*idx*, *vrblvl=0*)

Copies the system set in double precision to position idx in the systems pool.

polynomials.**copy_to_quad_double_syspool**(*idx*, *vrblvl=0*)

Copies the system set in quad double precision to position idx in the systems pool.

polynomials.**degree_of_double_polynomial**(*idx*, *vrblvl=0*)

Returns the degree of the polynomial with double precision coefficients stored at position idx.

polynomials.**get_double_double_laurent_polynomial**(*idx*, *vrblvl=0*)

Returns the string representation of the Laurent polynomial in double double precision, at index idx.

polynomials.**get_double_double_laurent_system**(*vrblvl=0*)

Returns the string representation of the Laurent polynomials in double double precision.

polynomials.**get_double_double_number_laurent_terms**(*equ*, *vrblvl=0*)

Returns the number of terms in the equation with index equ, set as a Laurent system in double double precision.

polynomials.**get_double_double_number_terms**(*equ*, *vrblvl=0*)

Returns the number of terms in the equation with index equ, set as a polynomial system in double double precision.

polynomials.**get_double_double_polynomial**(*idx*, *vrblvl=0*)

Returns the string representation of the polynomial in double double precision, at index idx.

polynomials.**get_double_double_system**(*vrblvl=0*)

Returns the string representation of the polynomials in double double precision.

polynomials.**get_double_laurent_polynomial**(*idx*, *vrblvl=0*)

Returns the string representation of the Laurent polynomial in double precision, at index idx.

polynomials.**get_double_laurent_system**(*vrblvl=0*)

Returns the string representation of the Laurent polynomials in double precision.

polynomials.**get_double_number_laurent_terms**(*equ*, *vrblvl=0*)

Returns the number of terms in the equation with index equ, set as a Laurent system in double precision.

polynomials.**get_double_number_terms**(*equ*, *vrblvl=0*)

Returns the number of terms in the equation with index equ, set as a polynomial system in double precision.

polynomials.**get_double_polynomial**(*idx*, *vrblvl=0*)

Returns the string representation of the polynomial in double precision, at index idx.

polynomials.**get_double_system**(*vrblvl=0*)

Returns the string representation of the polynomials in double precision.

polynomials.**get_quad_double_laurent_polynomial**(*idx*, *vrblvl=0*)

Returns the string representation of the Laurent polynomial in quad double precision, at index idx.

polynomials.**get_quad_double_laurent_system**(*vrblvl=0*)

Returns the string representation of the Laurent polynomials in quad double precision.

polynomials.**get_quad_double_number_laurent_terms**(*equ*, *vrblvl=0*)

Returns the number of terms in the equation with index equ, set as a Laurent system in quad double precision.

polynomials.**get_quad_double_number_terms**(*equ*, *vrblvl=0*)

Returns the number of terms in the equation with index equ, set as a polynomial system in quad double precision.

polynomials.**get_quad_double_polynomial**(*idx*, *vrblvl=0*)

Returns the string representation of the polynomial in quad double precision, at index idx.

polynomials.**get_quad_double_system**(*vrblvl=0*)

Returns the string representation of the polynomials in quad double precision.

polynomials.**initialize_double_double_syspool**(*dim*, *vrblvl=0*)

Initialize the systems pool in double double precision with dim.

polynomials.**initialize_double_syspool**(*dim*, *vrblvl=0*)

Initialize the systems pool in double precision with dim.

polynomials.**initialize_quad_double_syspool**(*dim*, *vrblvl=0*)

Initialize the systems pool in quad double precision with dim.

polynomials.**is_square**(*pols*, *vrblvl=0*)

Given in the list *pols* are string representations of Laurent polynomials. A system is square if it has as many unknowns as equations. Returns True if the system is square, False otherwise.

polynomials.**main**()

Runs tests on the set/get polynomials.

polynomials.**number_of_symbols**(*pols*, *vrblvl=0*)

Returns the number of symbols that appear as variables in the polynomials, givein in the list of strings pols. Useful to determine whether a system is square or not.

polynomials.**set_double_double_laurent_polynomial**(*idx*, *nvr*, *pol*, *vrblvl=0*)

Sets the polynomial by the string pol, in double double precision, with a number of variables no more than nvr, at index idx. The index starts at one, not at zero. This function does not set the dimension, which must be set to a value at least idx.

polynomials.**set_double_double_laurent_system**(*nvr*, *pols*, *vrblvl=0*)

Sets the laurent system defines by the strings in pols, in double double precision, with a number of variables no more than nvr. The dimension of the system is set to len(pols). Returns the sum of the return values of set_double_double_laurent_polynomial.

polynomials.**set_double_double_polynomial**(*idx*, *nvr*, *pol*, *vrblvl=0*)

Sets the polynomial by the string pol, in double double precision, with a number of variables no more than nvr, at index idx. The index starts at one, not at zero. This function does not set the dimension, which must be set to a value at least idx.

polynomials.**set_double_double_system**(*nvr*, *pols*, *vrblvl=0*)

Sets the system defines by the strings in pols, in double double precision, with a number of variables no more than nvr. The dimension of the system is set to len(pols). Returns the sum of the return values of set_double_double_polynomial.

polynomials.**set_double_laurent_polynomial**(*idx*, *nvr*, *pol*, *vrblvl=0*)

Sets the polynomial by the string pol, in double precision, with a number of variables no more than nvr, at index idx. The index starts at one, not at zero. This function does not set the dimension, which must be set to a value at least idx.

polynomials.**set_double_laurent_system**(*nvr*, *pols*, *vrblvl=0*)

Sets the laurent system defines by the strings in pols, in double precision, with a number of variables no more than nvr. The dimension of the system is set to len(pols). Returns the sum of the return values of set_double_laurent_polynomial.

polynomials.**set_double_polynomial**(*idx*, *nvr*, *pol*, *vrblvl=0*)

Sets the polynomial by the string pol, in double precision, with a number of variables no more than nvr, at index idx. The index starts at one, not at zero. This function does not set the dimension, which must be set to a value at least idx.

polynomials.**set_double_system**(*nvr*, *pols*, *vrblvl=0*)

Sets the system defines by the strings in pols, in double precision, with a number of variables no more than nvr. The dimension of the system is set to len(pols). Returns the sum of the return values of set_double_polynomial.

polynomials.**set_quad_double_laurent_polynomial**(*idx*, *nvr*, *pol*, *vrblvl=0*)

Sets the polynomial by the string pol, in quad double precision, with a number of variables no more than nvr, at index idx. The index starts at one, not at zero. This function does not set the dimension, which must be set to a value at least idx.

polynomials.**set_quad_double_laurent_system**(*nvr*, *pols*, *vrblvl=0*)

Sets the laurent system defines by the strings in pols, in quad double precision, with a number of variables no more than nvr. The dimension of the system is set to len(pols). Returns the sum of the return values of set_quad_double_polynomial.

polynomials.**set_quad_double_polynomial**(*idx*, *nvr*, *pol*, *vrblvl=0*)

Sets the polynomial by the string pol, in quad double precision, with a number of variables no more than nvr, at index idx. The index starts at one, not at zero. This function does not set the dimension, which must be set to a value at least idx.

polynomials.**set_quad_double_system**(*nvr*, *pols*, *vrblvl=0*)

Sets the system defines by the strings in pols, in quad double precision, with a number of variables no more than nvr. The dimension of the system is set to len(pols). Returns the sum of the return values of set_quad_double_polynomial.

polynomials.**size_double_double_syspool**(*vrblvl=0*)

Returns the size of the systems pool in double double precision.

polynomials.**size_double_syspool**(*vrblvl=0*)

Returns the size of the systems pool in double precision.

polynomials.**size_quad_double_syspool**(*vrblvl=0*)

Returns the size of the systems pool in quad double precision.

polynomials.**string_of_symbols**(*maxlen=100*, *vrblvl=0*)

Returns the list of all symbols (as strings), defined when storing polynomials. The maxlen on entry equals the maximum number of characters in the symbol string, that is: the sequence of all string representations of the symbols, separated by one space.

polynomials.**test_degree_of_double_polynomial**(*vrblvl=0*)

Tests the degree of a polynomial in double precision.

polynomials.**test_double_double_laurent_polynomial**(*vrblvl=0*)

Tests the setting and getting of a Laurent polynomial, in double double precision. The verbose level is given by vrblvl.

polynomials.**test_double_double_laurent_system**(*vrblvl=0*)

Tests the setting and getting of a laurent system, in double double precision. The verbose level is given by vrblvl.

polynomials.**test_double_double_number_laurent_terms**(*vrblvl=0*)

Tests the number of terms in a Laurent system set in double double precision.

polynomials.**test_double_double_number_terms**(*vrblvl=0*)

Tests the number of terms in a polynomial system set in double double precision.

polynomials.**test_double_double_polynomial**(*vrblvl=0*)

Tests the setting and getting of a polynomial, in double double precision. The verbose level is given by vrblvl.

polynomials.**test_double_double_syspool**(*vrblvl=0*)

Tests the systems pool in double double precision. The verbose level is given by vrblvl.

polynomials.**test_double_double_system**(*vrblvl=0*)

Tests the setting and getting of a system, in double double precision. The verbose level is given by vrblvl.

polynomials.**test_double_laurent_polynomial**(*vrblvl=0*)

Tests the setting and getting of a Laurent polynomial, in double precision. The verbose level is given by vrblvl.

polynomials.**test_double_laurent_system**(*vrblvl=0*)

Tests the setting and getting of a laurent system, in double precision. The verbose level is given by vrblvl.

polynomials.**test_double_number_laurent_terms**(*vrblvl=0*)

Tests the number of terms in a Laurent system set in double precision.

polynomials.**test_double_number_terms**(*vrblvl=0*)

Tests the number of terms in a polynomial system set in double precision.

polynomials.**test_double_polynomial**(*vrblvl=0*)

Tests the setting and getting of a polynomial, in double precision. The verbose level is given by vrblvl.

polynomials.**test_double_syspool**(*vrblvl=0*)

Tests the systems pool in double precision. The verbose level is given by vrblvl.

polynomials.**test_double_system**(*vrblvl=0*)

Tests the setting and getting of a system, in double precision. The verbose level is given by vrblvl.

polynomials.**test_is_square**(*vrblvl=0*)

Tests on catching an omitted comma between the polynomials.

polynomials.**test_quad_double_laurent_polynomial**(*vrblvl=0*)

Tests the setting and getting of a Laurent polynomial, in quad double precision. The verbose level is given by vrblvl.

polynomials.**test_quad_double_laurent_system**(*vrblvl=0*)

Tests the setting and getting of a laurent system, in quad double precision. The verbose level is given by vrblvl.

polynomials.**test_quad_double_number_laurent_terms**(*vrblvl=0*)

Tests the number of terms in a Laurent system set in quad double precision.

polynomials.**test_quad_double_number_terms**(*vrblvl=0*)

Tests the number of terms in a polynomial system set in quad double precision.

polynomials.**test_quad_double_polynomial**(*vrblvl=0*)

Tests the setting and getting of a polynomial, in quad double precision. The verbose level is given by vrblvl.

polynomials.**test_quad_double_syspool**(*vrblvl=0*)

Tests the systems pool in quad double precision. The verbose level is given by vrblvl.

polynomials.**test_quad_double_system**(*vrblvl=0*)

Tests the setting and getting of a system, in quad double precision. The verbose level is given by vrblvl.

4.2.3 functions in the module solutions

The documentation strings of the functions exported by the module `solutions` are listed below.

Exports functions on solutions.

class solutions.**DoubleSolution**(*sol*)

Wraps the functions on solution strings.

coordinates()

Returns the values of the coordinates of the solution, as a tuple of variable names and corresponding values.

diagnostics()

Returns the numerical diagnostics.

dictionary()

Returns the dictionary format of the solution.

multiplicity()

Returns the multiplicity.

numerals()

Returns the numerical values of the coordinates.

timevalue()

Returns the value of the continuation parameter.

variables()

Returns the variable names of the coordinates.

solutions.**append_double_double_solution_string**(*nvr*, *sol*, *vrblvl=0*)

Appends the string in sol to the solutions in double double precision, where the number of variables equals nvr.

solutions.**append_double_solution_string**(*nvr*, *sol*, *vrblvl=0*)

Appends the string in sol to the solutions in double precision, where the number of variables equals nvr.

solutions.**append_quad_double_solution_string**(*nvr*, *sol*, *vrblvl=0*)

Appends the string in sol to the solutions in quad double precision, where the number of variables equals nvr.

solutions.**clear_double_double_solutions**(*vrblvl=0*)

Clears the solutions defined in double double precision.

solutions.**clear_double_solutions**(*vrblvl=0*)

Clears the solutions defined in double precision.

solutions.**clear_quad_double_solutions**(*vrblvl=0*)

Clears the solutions defined in quad double precision.

solutions.**condition_tables**(*sols*, *vrblvl=0*)

The input in sols is a list of PHCpack string solutions. A condition table is triplet of three frequency tables, computed from the diagnostics (err, rco, res) of each solution. The i-th entry in each frequency table counts the number of doubles x which floor(-log10(x)) mapped to the index i. Small numbers are mapped to the right of the table, large numbers are mapped to the left of the table.

solutions.**coordinates**(*sol*, *vrblvl=0*)

Returns the coordinates of the solution in the PHCpack solution string *sol*, as a tuple of two lists: (names, values). The list names contains the strings of the variable names. The list values contains the complex values for the coordinates of the solution. The entries in the list names correspond to the entries in the list values.

solutions.**diagnostics**(*sol*, *vrblvl=0*)

Extracts the diagnostics (err, rco, res) from the PHCpack string solution in *sol* and returns a triplet of three floats.

solutions.**endmultiplicity**(*sol*, *vrblvl=0*)

Returns the value of t at the end and the multiplicity as (t,m) for the PHCpack solution string *sol*.

solutions.**evaluate**(*pols*, *dsol*, *vrblvl=0*)

Evaluates a list of polynomials given as string in *pols* at the solution in dictionary format in *dsol*.

solutions.**evaluate_polynomial**(*pol*, *dsol*, *vrblvl=0*)

Evaluates the polynomial *pol* at the solution dictionary *dsol* by string substitution.

solutions.**filter_real**(*sols*, *tol*, *oper*, *vrblvl=0*)

Filters the real solutions in *sols*. The input parameters are

1. *sols* is a list of solution strings in PHCpack format,
2. *tol* is the tolerance on the absolute value of the imaginary parts of the coordinates of the solution.
3. *oper* is either 'select' or 'remove'

   if *oper* == 'select' then solutions that are considered real are selected and in the list on return,

   if *oper* == 'remove' then solutions that are considered real are in the list on return.

solutions.**filter_regular**(*sols*, *tol*, *oper*, *vrblvl=0*)

Filters solutions in *sols* for the estimate of the inverse of the condition number. The input parameters are

1. *sols* is a list of solution strings in PHCpack format,
2. *tol* is the tolerance on the value for the estimate rco for the inverse of the condition number to decide whether a solution is singular (if rco < *tol*) or not.
3. *oper* is either 'select' or 'remove'

   if *oper* == 'select' then solutions with value rco > *tol* are selected and in the list on return,

   if *oper* == 'remove' then solutions with value rco <= *tol* are in the list on return.

solutions.**filter_vanishing**(*sols*, *tol*, *vrblvl=0*)

Returns the list of solutions in *sols* that have a residual less than or equal to the given tolerance in *tol*.

solutions.**filter_zero_coordinates**(*sols*, *varname*, *tol*, *oper*, *vrblvl=0*)

Filters the solutions in *sols* for variables that have a value less than the tolerance. The input parameters are

1. *sols* is a list of solution strings in PHCpack format,
2. *varname* is a string with the name of the variable,
3. *tol* is the tolerance to decide whether a complex number equals zero or not, and
4. *oper* is either 'select' or 'remove'

   if *oper* == 'select' then solutions with value for the variable v that is less than *tol* are selected and in the list on return,

   if *oper* == 'remove' then solutions with value for the variable v that is less than *tol* are removed and not in the list on return.

solutions.**formdictlist**(*sols*, *precision='d'*, *vrblvl=0*)

Given in *sols* is a list of strings. Each string in *sols* represents a solution, in the PHCpack format. On return is the list of dictionaries. Each dictionary in the list of return stores each solution of the list *sols* in the dictionary format. By default, the precision of the coordinates is assumed to be double float ('d' on input). If the precision is not 'd', then the coordinates of the solution are returned as Python complex number string representations.

solutions.**get_double_double_solutions**(*vrblvl=0*)

Returns the solution strings in double double precision.

solutions.**get_double_solutions**(*vrblvl=0*)

Returns the solution strings in double precision.

solutions.**get_next_double_double_solution**(*idx*, *vrblvl=0*)

Returns the string representation of the next solution in double double precision, at the index idx. The vrblvl is the verbose level.

solutions.**get_next_double_solution**(*idx*, *vrblvl=0*)

Returns the string representation of the next solution in double precision, at the index idx. The vrblvl is the verbose level.

solutions.**get_next_quad_double_solution**(*idx*, *vrblvl=0*)

Returns the string representation of the next solution in quad double precision, at the index idx. The vrblvl is the verbose level.

solutions.**get_quad_double_solutions**(*vrblvl=0*)

Returns the solution strings in quad double precision.

solutions.**is_real**(*sol*, *tol*, *vrblvl=0*)

Returns True if the solution in *sol* is real with respect to the given tolerance *tol*: if the absolute value of the imaginary part of all coordinates are less than *tol*.

solutions.**is_vanishing**(*sol*, *tol*, *vrblvl=0*)

Given in *sol* is a solution string and *tol* is the tolerance on the residual. Returns True if the residual of *sol* is less than or equal to *tol*. Returns False otherwise.

solutions.**main**()

Runs some tests on solutions.

solutions.**make_solution**(*names*, *values*, *err=0.0*, *rco=1.0*, *res=0.0*, *tval=0*, *multiplicity=1*, *vrblvl=0*)

Makes the string representation in PHCpack format with in *names* a list of strings for the variables names and in *values* a list of (complex) values for the coordinates. For example:

s = make_solution(['x','y'],[(1+2j), 3])

returns the string s to represent the solution with coordinates (1+2j) and 3 for the variables x and y. The imaginary unit is the Python j instead of i. Other arguments for this function are

1. *err* is the magnitude of an update, or the forward error,
2. *rco* is the estimate for the inverse condition number,
3. *res* is the value of the residual, or backward error,
4. *tval* is the value for the continuation parameter t,
5. *multiplicity* is the multiplicity of the solution.

For those above arguments, default values are provided. Applying the function coordinates on the result of make_solution returns the tuple of arguments given on input to **make_solution()**.

solutions.**map_double**(*freqtab*, *nbr*, *vrblvl=0*)

On input in freqtab is a list of integer numbers and nbr is a double. The list freqtab represents a frequency table of magnitudes. The magnitude of the double nbr is mapped into the frequency table. The counter in freqtab that will be updated is at position floor(-log10(nbr))

solutions.**move_double_double_solution_cursor**(*idx*, *vrblvl=0*)

Moves the cursor to the next solution, following the index, in double double precision.

solutions.**move_double_solution_cursor**(*idx*, *vrblvl=0*)

Moves the cursor to the next solution, following the index, in double precision.

solutions.**move_quad_double_solution_cursor**(*idx*, *vrblvl=0*)

Moves the cursor to the next solution, following the index, in quad double precision.

solutions.**number_double_double_solutions**(*vrblvl=0*)

Returns the number of solutions in double double precision. The vrblvl is the verbose level.

solutions.**number_double_solutions**(*vrblvl=0*)

Returns the number of solutions in double precision. The vrblvl is the verbose level.

solutions.**number_quad_double_solutions**(*vrblvl=0*)

Returns the number of solutions in quad double precision. The vrblvl is the verbose level.

solutions.**numerals**(*dsol*, *vrblvl=0*)

Given the dictionary format of a solution *dsol*, returns the list of numeric values of the variables in the solution.

solutions.**set_double_double_solutions**(*nvr*, *sols*, *vrblvl=0*)

Sets the solutions in double double precision, with the strings in sols, where the number of variables equals nvr.

solutions.**set_double_solutions**(*nvr*, *sols*, *vrblvl=0*)

Sets the solutions in double precision, with the strings in sols, where the number of variables equals nvr.

solutions.**set_quad_double_solutions**(*nvr*, *sols*, *vrblvl=0*)

Sets the solutions in quad double precision, with the strings in sols, where the number of variables equals nvr.

solutions.**str2complex**(*scn*, *vrblvl=0*)

The string *scn* contains a complex number, the real and imaginary part separated by spaces. On return is the Python complex number.

solutions.**string_complex**(*scn*, *vrblvl=0*)

The string *scn* contains a complex number, the real and imaginary part separated by spaces. On return is the string representation of a complex number, in Python format. The use of this function is for when the coordinates are calculated in higher precision.

solutions.**string_coordinates**(*sol*, *vrblvl=0*)

Returns the coordinates of the solution in the PHCpack solution string *sol*, as a tuple of two lists: (names, values). For each name in names there is a value in values. The list names contains the strings of the variable names. The list values contains the values of the coordinates, represented as strings. This function is useful for when the coordinates are computed in higher precision.

solutions.**strsol2dict**(*sol*, *precision='d'*, *vrblvl=0*)

Converts the solution in the string *sol* into a dictionary format. By default, the precision of the coordinates is assumed to be double float ('d' on input). If the precision is not 'd', then the coordinates of the solution are returned as Python complex number string representations.

solutions.**test_double_functions**(*vrblvl=0*)

Generates a random trinomial system, solves it, converts the solutions, and then sums the multiplicities. The verbose level is given by vrblvl.

solutions.**test_double_solution_class**(*vrblvl=0*)

Tests the methods in the class DoubleSolution. The verbose level is given by vrblvl.

solutions.**variables**(*dsol*, *vrblvl=0*)

Given the dictionary format of a solution in *dsol*, returns the list of variables.

solutions.**verify**(*pols*, *sols*, *vrblvl=0*)

Verifies whether the solutions in *sols* satisfy the polynomials of the system in *pols*. Returns the sum of the absolute values of the residuals in all polynomials.

solutions.**write_double_double_solutions**(*vrblvl=0*)

Writes the solutions stored in double double precision.

solutions.**write_double_solutions**(*vrblvl=0*)

Writes the solutions stored in double precision.

solutions.**write_quad_double_solutions**(*vrblvl=0*)

Writes the solutions stored in quad double precision.

4.2.4 functions in the module volumes

The mixed volume of the tuple of Newton polytopes of a polynomial systems is a generically sharp root count on the number of isolated solutions with nonzero coordinates. For all affine solutions (also counting solutions with zero coordinates), the stable mixed volume provides a generically sharp root count.

Exports functions to compute mixed volumes and stable mixed volumes. The mixed volume of a polynomial system is a generically sharp upper bound on the number of isolated solutions with nonzero coordinates. Stable mixed volumes count solutions with zero coordinates as well. Polyhedral homotopies solve random coefficient systems, tracking exactly as many paths as the mixed volume.

volumes.**are_cells_stable**(*vrblvl=0*)

Returns True if stable mixed cells were computed, returns False otherwise.

volumes.**cell_mixed_volume**(*idx*, *vrblvl=0*)

Returns the mixed volume of cell with index idx.

volumes.**clear_cells**(*vrblvl=0*)

Clears the computed mixed cells.

volumes.**compute_mixed_volume**(*demics=True*, *vrblvl=0*)

Returns the mixed volume of the polynomial system, set in double precision. The demics flag indicates if dynamic enumeration as implemented by the software DEMiCs will be used, otherwise MixedVol is called. The verbose level is given by vrblvl.

volumes.**compute_stable_mixed_volume**(*demics=True*, *vrblvl=0*)

Returns the mixed and the stable mixed volume of the polynomial system set in double precision. The demics flag indicates if dynamic enumeration as implemented by the software DEMiCs will be used, otherwise MixedVol is called. The verbose level is given by vrblvl.

volumes.**double_double_polyhedral_homotopies**(*vrblvl=0*)

Runs the polyhedral homotopies and returns a random coefficient system, in double double precision arithmetic, based on the mixed volume computation

volumes.**double_double_random_coefficient_system**(*vrblvl=0*)

Makes a random coefficient system using the type of mixture and the supports used to compute the mixed volume. Sets the system in double double precision.

volumes.**double_double_solve_start_system**(*idx*, *vrblvl=0*)

Solves the start system corresponding to the cell with index idx, using double double precision arithmetic. Returns the number of solutions found, which must equal the mixed volume of the cell.

volumes.**double_double_track_path**(*cellidx*, *pathidx*, *vrblvl=0*)

Tracks path with index pathidx defined by cell with index cellidx. Prior to calling this function, the start system corresponding to the cell must have been solved, in double double precision.

volumes.**double_polyhedral_homotopies**(*vrblvl=0*)

Runs the polyhedral homotopies and returns a random coefficient system, in double precision arithmetic, based on the mixed volume computation

volumes.**double_random_coefficient_system**(*vrblvl=0*)

Makes a random coefficient system using the type of mixture and the supports used to compute the mixed volume. Sets the system in double precision.

volumes.**double_solve_start_system**(*idx*, *vrblvl=0*)

Solves the start system corresponding to the cell with index idx, using double precision arithmetic. Returns the number of solutions found, which must equal the mixed volume of the cell.

volumes.**double_track_path**(*cellidx*, *pathidx*, *vrblvl=0*)

Tracks path with index pathidx defined by cell with index cellidx. Prior to calling this function, the start system corresponding to the cell must have been solved, in double precision.

volumes.**main**()

Runs tests on mixed volumes and stable mixed volumes.

volumes.**make_random_coefficient_system**(*pols*, *demics=True*, *precision='d'*, *vrblvl=0*)

For a polynomial system in the list pols with as many equations as unknowns, computes the mixed volume (by default by DEMiCs) and runs the polyhedral homotopies to solve a random coefficient system in double, double double, or quad double precision, if the precision flag is set to 'd', 'dd', or 'qd' respectively. Returns a tuple with the mixed volume, random coefficient system, and the solutions of the random coefficient system.

volumes.**mixed_volume**(*pols*, *demics=True*, *vrblvl=0*)

Returns the mixed volume of the polynomial system in the list pols. The polynomial system must have as many equations as unknowns. The demics flag indicates if dynamic enumeration as implemented by the software DEMiCs will be used, otherwise MixedVol is called. The verbose level is given by vrblvl.

volumes.**number_of_cells**(*vrblvl=0*)

Returns the number of cells computed by the mixed_volume function.

volumes.**number_of_original_cells**(*vrblvl=0*)

Returns the number of original cells, the cells without artificial origin, computed by the mixed_volume function.

volumes.**number_of_stable_cells**(*vrblvl=0*)

Returns the number of stable cells computed by the mixed_volume function.

volumes.**quad_double_polyhedral_homotopies**(*vrblvl=0*)

Runs the polyhedral homotopies and returns a random coefficient system, in quad double precision arithmetic, based on the mixed volume computation

volumes.**quad_double_random_coefficient_system**(*vrblvl=0*)

Makes a random coefficient system using the type of mixture and the supports used to compute the mixed volume. Sets the system in quad double precision.

volumes.**quad_double_solve_start_system**(*idx*, *vrblvl=0*)

Solves the start system corresponding to the cell with index idx, using quad double precision arithmetic. Returns the number of solutions found, which must equal the mixed volume of the cell.

volumes.**quad_double_track_path**(*cellidx*, *pathidx*, *vrblvl=0*)

Tracks path with index pathidx defined by cell with index cellidx. Prior to calling this function, the start system corresponding to the cell must have been solved, in quad double precision.

volumes.**set_double_coefficient_system**(*vrblvl=0*)

Sets the constructed random coefficient system to the system in double precision.

volumes.**set_double_double_coefficient_system**(*vrblvl=0*)

Sets the constructed random coefficient system to the system in double double precision.

volumes.**set_double_double_polyhedral_homotopy**(*vrblvl=0*)

Sets the polyhedral homotopy in double double precision based on the lifted supports used to compute the mixed volume. Initializes the data to store the solutions.

volumes.**set_double_double_start_solution**(*cellidx*, *solidx*, *vrblvl=0*)

Sets the start solution of index solidx, with cell with index cellidx, to the solutions in double double precision.

volumes.**set_double_polyhedral_homotopy**(*vrblvl=0*)

Sets the polyhedral homotopy in double precision based on the lifted supports used to compute the mixed volume. Initializes the data to store the solutions.

volumes.**set_double_start_solution**(*cellidx*, *solidx*, *vrblvl=0*)

Sets the start solution of index solidx, with cell with index cellidx, to the solutions in double precision.

volumes.**set_quad_double_coefficient_system**(*vrblvl=0*)

Sets the constructed random coefficient system to the system in quad double precision.

volumes.**set_quad_double_polyhedral_homotopy**(*vrblvl=0*)

Sets the polyhedral homotopy in quad double precision based on the lifted supports used to compute the mixed volume. Initializes the data to store the solutions.

volumes.**set_quad_double_start_solution**(*cellidx*, *solidx*, *vrblvl=0*)

Sets the start solution of index solidx, with cell with index cellidx, to the solutions in quad double precision.

volumes.**stable_mixed_volume**(*pols*, *demics=True*, *vrblvl=0*)

Returns the mixed and the stable mixed volume of the polynomial system in the list pols. The polynomial system must have as many equations as unknowns. The demics flag indicates if dynamic enumeration as implemented by the software DEMiCs will be used, otherwise MixedVol is called. The verbose level is given by vrblvl.

volumes.**test_double_double_polyhedral_homotopies**(*vrblvl=0*)

Test the polyhedral homotopies in double double precision to compute and solve a random coefficient system. The verbose level is given by vrblvl.

volumes.**test_double_polyhedral_homotopies**(*vrblvl=0*)

Test the polyhedral homotopies in double precision to compute and solve a random coefficient system. The verbose level is given by vrblvl.

volumes.**test_make_random_coefficient_system**(*vrblvl=0*)

Test the making of a random coefficient system. The verbose level is given by vrblvl.

volumes.**test_mixed_volume**(*vrblvl=0*)

Computes the mixed volume of a simple example. The verbose level is given by vrblvl.

volumes.**test_quad_double_polyhedral_homotopies**(*vrblvl=0*)

Test the polyhedral homotopies in quad double precision to compute and solve a random coefficient system. The verbose level is given by vrblvl.

volumes.**test_stable_mixed_volume**(*vrblvl=0*)

Computes the stable mixed volume of a simple example. The verbose level is given by vrblvl.

4.2.5 functions in the module solver

The documentation strings of the functions exported by the module `solver` of the package phcpy are listed below.

Exports the blackbox solver for isolated solutions of square systems, in double, double double, and quad double precision. Also Laurent systems are accepted on input.

solver.**main**()

Runs tests on the blackbox solver.

solver.**random_trinomials**(*vrblvl=0*)

Returns a system of two trinomials equations for testing. A trinomial consists of three monomials in two variables. Exponents are uniform between 0 and 5 and coefficients are on the complex unit circle.

solver.**real_random_trinomials**(*sys*, *vrblvl=0*)

On input in sys are two random trinonials with complex coefficients, in the format what **random_trinomials()** returns. On return is a list of two real random trinomials with the same monomial structure but with random real coefficients in [-1,+1].

solver.**solve**(*pols*, *tasks=0*, *mvfocus=0*, *precision='d'*, *checkin=True*, *dictionary_output=False*, *vrblvl=0*)

Calls the blackbox solver to compute all isolated solutions. To compute all solutions, also all positive dimensional solution sets, with a numerical irreducible decomposition, see phcpy.decomposition. On input in *pols* is a list of strings. The number of tasks for multithreading is given by *tasks*. The default zero value for *tasks* indicates no multithreading. If the *mvfocus* option is set to one, then only mixed volumes and polyhedral homotopies will be applied in the solver, and no degree bounds will be computed, as is already the case when the input system is genuinely laurent and has negative exponents. Three levels of precision are supported:

d: standard double precision (1.1e-15 or 2^(-53)),

dd: double double precision (4.9e-32 or 2^(-104)),

qd: quad double precision (1.2e-63 or 2^(-209)).

If *checkin* (by default), the input *pols* is checked for being square. If *dictionary_output*, then on return is a list of dictionaries, else the returned list is a list of strings. If *vrblvl* is larger than 0, then the names of the procedures called in the running of the blackbox solver will be listed.

solver.**solve_checkin**(*pols*, *msg*, *vrblvl=0*)

Checks whether the system defined by the list of strings in *pols* is square. If so, True is returned. Otherwise, the error message in the string *msg* is printed to help the user.

solver.**solve_double_double_laurent_system**(*nbtasks=0*, *mvfocus=0*, *vrblvl=0*)

Solves the laurent system stored in double double precision, where nbtasks equals the number of tasks, no multitasking if zero, mvfocus equals zero by default and all root counts are computed, otherwise, the focus is on mixed volumes and polyhedral homotopies, and vrblvl is the verbose level.

solver.**solve_double_double_system**(*nbtasks=0*, *mvfocus=0*, *vrblvl=0*)

Solves the system stored in double double precision, where nbtasks equals the number of tasks, no multitasking if zero, mvfocus equals zero by default and all root counts are computed, otherwise, the focus is on mixed volumes and polyhedral homotopies, and vrblvl is the verbose level.

solver.**solve_double_laurent_system**(*nbtasks=0*, *mvfocus=0*, *vrblvl=0*)

Solves the laurent system stored in double precision, where nbtasks equals the number of tasks, no multitasking if zero, mvfocus equals zero by default and all root counts are computed, otherwise, the focus is on mixed volumes and polyhedral homotopies, and vrblvl is the verbose level.

solver.**solve_double_system**(*nbtasks=0*, *mvfocus=0*, *vrblvl=0*)

Solves the system stored in double precision, where nbtasks equals the number of tasks, no multitasking if zero, mvfocus equals zero by default and all root counts are computed, otherwise, the focus is on mixed volumes and polyhedral homotopies, and vrblvl is the verbose level.

solver.**solve_quad_double_laurent_system**(*nbtasks=0*, *mvfocus=0*, *vrblvl=0*)

Solves the laurent system stored in quad double precision, where nbtasks equals the number of tasks, no multitasking if zero, mvfocus equals zero by default and all root counts are computed, otherwise, the focus is on mixed volumes and polyhedral homotopies, and vrblvl is the verbose level.

solver.**solve_quad_double_system**(*nbtasks=0*, *mvfocus=0*, *vrblvl=0*)

Solves the system stored in quad double precision, where nbtasks equals the number of tasks, no multitasking if zero, mvfocus equals zero by default and all root counts are computed, otherwise, the focus is on mixed volumes and polyhedral homotopies, and vrblvl is the verbose level.

solver.**test_double_double_laurent_solve**(*vrblvl=0*)

Solves a simple laurent system in double double precision. The verbose level is given by vrblvl.

solver.**test_double_double_solve**(*vrblvl=0*)

Solves a simple system in double double precision. The verbose level is given by vrblvl.

solver.**test_double_laurent_solve**(*vrblvl=0*)

Solves a simple laurent system in double precision. The verbose level is given by vrblvl.

solver.**test_double_solve**(*vrblvl=0*)

Solves a simple system in double precision. The verbose level is given by vrblvl.

solver.**test_quad_double_laurent_solve**(*vrblvl=0*)

Solves a simple laurent system in quad double precision. The verbose level is given by vrblvl.

solver.**test_quad_double_solve**(*vrblvl=0*)

Solves a simple system in quad double precision. The verbose level is given by vrblvl.

solver.**test_solve**(*vrblvl=0*)

Tests the solve function on a simple system.

solver.**test_trinomial_solve**(*vrblvl=0*)

Generates a random trinomial system and solves it. The verbose level is given by vrblvl.

4.2.6 functions in the module examples

To demonstrate the capabilities of the solver and the relevance of polynomial systems to various fields of science and engineering, several examples from the literature are provided by the `examples` module.

This module offers functions to returns the lists of polynomial strings of some well known examples. The test solves all systems and tests on the number of solutions. Running the tests can take some time.

examples.**binomials**()

A binomial system is a polynomial system where every equation has exactly two monomials with nonzero coefficient. A pure dimensional binomial system can be solved quickly. The example below has negative exponents.

examples.**cyclic7**()

The cyclic 7-roots problem is a standard benchmark problem for polynomial system solvers. Reference: Backelin, J. and Froeberg, R.: How we proved that there are exactly 924 cyclic 7-roots, Proceedings of ISSAC-91, pp 103-111, ACM, New York, 1991.

examples.**fbrfive4**()

Four-bar linkage through five points, 4-dimensional version. Reference: Charles W. Wampler: Isotropic coordinates, circularity and Bezout numbers: planar kinematics from a new perspective. Publication R&D-8188, General Motors Corporation, Research and Development Center. Proceedings of the 1996 ASME Design Engineering Technical Conference, Irvine, California August 18-22, 1996. CD-ROM edited by McCarthy, J.M., American society of mechanical engineers.

examples.**game4two**()

Totally mixed Nash equilibria for 4 players with two pure strategies. See the paper by Andrew McLennan on The maximal generic number of pure Nash equilibria, Journal of Economic Theory, Vol 72, pages 408-410, 1997. Another reference is the paper by Richard D. McKelvey and Andrew McLennan: on The maximal number of regular totally mixed Nash equilibria, Journal of Economic Theory, Volume 72, pages 411-425, 1997. For more on equilibria, see David M. Kreps and Robert Wilson: Sequential Equibria, Econometrica, Vol 50, No 4, pages 863-894,1982.

examples.**katsura6**()

A problem of magnetism in physics, see for examples the paper by S. Katsura, in New Trends in Magnetism, edited by M.D. Coutinho-Filho and S.M. Resende, World Scientific, 1990. It became part of the PoSSo test suite, Shigotoshi Katsura: "Users posing problems to PoSSo", in the PoSSo Newsletter, no. 2, July 1994, edited by L. Gonzalez-Vega and T. Recio. See also the paper by W. Boege, R. Gebauer, and H. Kredel: Some examples for solving systems of algebraic equations by calculating Groebner bases, J. Symbolic Computation, 2:83-98, 1986. The system can be written for any dimension.

examples.**main**()

Solves the systems and checks on their number of solutions.

examples.**noon3**()

A neural network modeled by an adaptive Lotka-Volterra system. See the paper by Karin Gatermann on Symbolic solution of polynomial equation systems with symmetry, Proceedings of ISSAC-90, pages 112-119, ACM New York, 1990. The system belongs to a family of problems of V.W. Noonburg: A neural network modeled by an adaptive Lotka-Volterra system, SIAM J. Appl. Math., Vol. 49, No. 6, 1779-1792, 1989.

examples.**rps10**()

RPS Serial Chains for 10 positions on a circular hyperboloid. From the paper of Hai-Jun Su and J. Michael McCarthy: Kinematics Synthesis of RPS Serial Chains, In Proceedings of the ASME Design Engineering Technical Conferences (CDROM). Paper DETC03/DAC-48813. Chicago, IL, Sept.02-06, 2003.

examples.**sevenbar**()

Returns the list of strings which represent Laurent polynomials for a special sevenbar mechanism that has isolated solutions and a cubic curve. A reference for the general case is the paper by Carlo Innocenti: Polynomial solution to the position analysis of the 7-line Assur kinematic chain with one quaternary link, in Mech. Mach. Theory, Vol. 30, No. 8, pages 1295-1303, 1995. The special case was introduced in the paper with title: Numerical decomposition of the solution sets of polynomial systems into irreducible components, SIAM J. Numer. Anal. 38(6):2022-2046, 2001, by Andrew Sommese, Jan Verschelde, and Charles Wampler.

examples.**solve_binomials**()

Runs the test on solving the binomials example. Checks that the number of solutions equals 20. Returns 0 if if the test passed, 1 if the test failed.

examples.**solve_cyclic7**()

Runs the test on solving the cyclic 7-roots problem. Checks that the number of solutions equals 924. Returns 0 if the test passed, 1 if the test failed.

examples.**solve_fbrfive4**()

Runs the test on solving a generic 4-bar problem. Checks that the number of solutions equals 36. Returns 0 if the test passed, 1 if the test failed.

examples.**solve_game4two**()

Runs the test on solving a Nash equilibrium problem. Checks that the number of solutions equals 9. Returns 0 if the test passed, 1 if the test failed.

examples.**solve_katsura6**()

Runs the test on solving the katsura6 problem. Checks that the number of solutions equals 64. Returns 0 if the test passed, 1 if the test failed.

examples.**solve_noon3**()

Test on solving the noon3 system. Checks that the number of solutions equals 21. Returns 0 if the test passed, 1 if the test failed.

examples.**solve_rps10**()

Test on solving a mechanical design problem. Checks that the number of solutions equals 1024. Returns 0 if the test passed, 1 if the test failed.

examples.**solve_sevenbar**()

Test on solving the sevenbar problem. Checks that the degree of the curve is one and that there are six isolated solutions.

examples.**solve_stewgou40**()

Test on solving a fully reall Stewart-Gough platform. Checks that the number of solutions equals 40. Returns 0 if the test passed, 1 if the test failed.

examples.**solve_sysd1**()

Runs the test on solving the benchmark problem D1. Checks that the number of solutions equals 48. Returns 0 if the test passed, 1 if the test failed.

examples.**solve_tangents**()

Test on solving the tangents to 4 spheres problem. Checks that the number of solutions equals 6. Returns 0 if the test passed, 1 if the test failed.

examples.**stewgou40**()

Stewart-Gough platform with 40 real postures. From the paper of Peter Dietmaier: The Stewart-Gough platform of general geometry can have 40 real postures, pages 1-10, in Advances in Robot Kinematics: Analysis and Control, edited by Jadran Lenarcic and Manfred L. Husty, Kluwer Academic Publishers, Dordrecht, 1998.

examples.**sysd1**()

A sparse system, know as benchmark D1 of the paper by H. Hong and V. Stahl: Safe Starting Regions by Fixed Points and Tightening, Computing 53(3-4): 322-335, 1995.

examples.**tangents**()

Three lines tangent to four given spheres, each with multiplicity 4. Originally formulated as a polynomial system by Cassiano Durand. Thorsten Teobald came up with the positioning of the centers of the spheres, each with radius 0.5 at the vertices of a tethrahedron. See the paper by Frank Sottile: From Enumerative Geometry to Solving Systems of Polynomial Equations". In "Computations in Algebraic Geometry with Macaulay 2", edited by David Eisenbud, Daniel R. Grayson, Michael Stillman, and Bernd Sturmfels, Volume 8 of Algorithms and Computation in Mathematics, pages 101-129, Springer-Verlag, 2002. Another reference is the paper by Frank Sottile and Thorsten Theobald: Lines tangents to 2n - 2 spheres in R^n. Trans. Amer. Math. Soc. 354:4815-4829, 2002.

4.2.7 functions in the module families

Polynomial system are often defined for any dimension that is: for any number of equations and number of variables. Such families of polynomial systems are important benchmark systems, for example: the cyclic n-roots system.

The module families contains scripts to generate polynomial systems for any dimension.

families.**adjacent_minors**(*rows*, *cols*)

Returns all adjacent 2-by-2 minors of a general matrix of dimensions rows by cols. This system originated in a paper on lattice walks and primary decomposition, written by P. Diaconis, D. Eisenbud, and B. Sturmfels, published by Birkhauser in 1998 in Mathematical Essays in Honor of Gian-Carlo Rota, edited by B. E. Sagan and R. P. Stanley, volume 161 of Progress in Mathematics, pages 173–193. See also the paper by S. Hosten and J. Shapiro on Primary decomposition of lattice basis ideals, published in 2000 in the Journal of Symbolic Computation, volume 29, pages 625-639.

families.**chandra**(*dim*, *par=0.51234*)

Generates the equations of the Chandrasekhar H-equation for the given dimension dim and parameter par. The name of the problem stems from the 1960 Dover publication Radiative Transfer by S. Chandrasekhar. The problem was used as an illustration in the paper by C.T. Kelley on Solution of the Chandrasekhar H-equation by Newton's method, published in J. Math. Phys. 21, pages 1625-1628, 1980. It featured in the paper by

Jorge J. More on A collection of nonlinear model problems, published in volume 26 of the Lectures in Applied Mathematics, pages 723-762, AMS 1990 and in the paper by Laureano Gonzalez-Vega on Some examples on problem solving by using the symbolic viewpoint when dealing with polynomial systems of equations, published in Computer Algebra in Science and Engineering, pages 102-116, World Scientific, 1995.

families.**cyclic**(*dim*)

Returns a list of string representing the polynomials of the cyclic n-roots system. This system entered the computer algebra literature in a technical report by J. Davenport on Looking at a set of equations, published in 1987 as Bath Computer Science Technical Report 87-06. Another technical report by J. Backelin in 1989 has the title Square multiples n give infinitely many cyclic n-roots, publised as Reports, Matematiska Institutionen 8, Stockholms universitet. Another interesting preprint is written by U. Haagerup, available at http://www.math.ku.dk/~haagerup, on cyclic p-roots of prime length p and related complex Hadamard matrices.

families.**firsteqs**(*dim*)

Returns the list of equations defining the relations between the S[i,j] and the r[i,j] variables, for all i < j, for i from 1 to dim-1. Since the S[i,j] variables occur linearly, with these equations we can rewrite S[i,j] in terms of the corresponding r[i,j] variables. The elimination of the S[i,j] comes at the expense of high degrees in the r[i,j] variables of the remaining equations.

families.**generic_nash_system**(*nbplayers*)

Returns a list of strings representing polynomials that define totally mixed Nash equilibria for a number of players equals to nbplayers with two pure strategies. The problem setup is generic in the sense that the utilities are uniformly generated positive floats in [0,1]. For n players, the n-homogeneous Bezout number provides a generically exact count on the number of equilibria, see the paper by Richard D. McKelvey and Andrew McLennan on the maximal number of regular totally mixed Nash equilibria, published in the Journal of Economic Theory, volume 72, pages 411-425, 1997.

families.**indeterminate_matrix**(*rows*, *cols*)

Returns a list of lists with as many lists as the value of rows. Each rows has as many indeterminates as the value of cols. The lists of lists contains the data for a matrix of dimension rows by cols of variables.

families.**katsura**(*dim*)

Returns the list of strings to represent the system of Katsura. The system originated in a paper by S. Katsura on Spin glass problem by the method of integral equation of the effective field, published in 1990 by World Scientific in the volume New Trends in Magnetism, edited by M. D. Coutinho-Filho and S. M. Resende, pages 110-121. The note by S. Katsura on Users posing problems to PoSSo appeared in 1994 in the second number of the PoSSo Newsletter, edited by L. Gonzalez-Vega and T. Recio.

families.**katsura_variable**(*var*, *dim*)

Returns the variable U(var, dim) for use in the function katsura.

families.**nash**(*nbplayers*, *player*)

Returns the string representation of one equation for a player to compute the totally mixed Nash equibria for nbplayers with two pure strategies, with random positive utilities.

families.**nbodyeqs**(*dim*, *mas*)

The central configurations of the n-body problem can be defined via the Albouy-Chenciner equations, by A. Albouy and A. Chenciner: Le probleme des n corps et les distances mutuelles. Inv. Math. 131, 151-184, 1998; and the paper by M. Hampton and R. Moeckel on Finiteness of relative equilibria of the four-body problem. Inv. Math. 163, 289-312, 2006. Returns a list of strings, representing the central configurations for the n-body problem, where n = dim and with masses in the list mas. We require that len(mas) == dim.

families.**noon**(*dim*, *parameter=1.1*)

Returns the list of strings to represent the system of Noonburg. The system originates in a paper by V. W. Noonburg on a neural network modeled by an adaptive Lotka-Volterra system, published in 1989 in volume

49 of SIAM Journal on Applied Mathematics, pages 1779-1792. It appeared also in a paper by K. Gatermann entitled Symbolic solution of polynomial equation systems with symmetry, published by ACM in 1990 in the proceedings of ISSAC-90, pages 112-119.

families.**pieri_problem**(*mdim*, *pdim*, *real=True*)

Returns a system that expresses the intersection of pdim-planes with mdim*pdim general mdim-planes in (mdim+pdim)-space. When real is True, the generated mdim-planes are osculating a rational normal curve and all solutions are expected to be real. If real is False, then random complex planes are generated. For reality of solutions of polynomial systems, see the book by Frank Sottile: Real Solutions to Equations from Geometry, volume 57 of University Lecture Series, AMS, 2011.

families.**poleqs**(*dim*, *masses*)

Returns the list of polynomial equations for the central configurations, for as many masses as the dimension dim.

families.**recpol**(*nbplayers*, *player*, *ind*, *acc*)

Recursive generation of one polynomial, called by the function nash below.

families.**strvar**(*name*, *i*, *j*)

Returns the string representation for the variable with the given name and indices i and j, i != j. Swaps the values for i and j if i > j.

families.**test**()

Writes particular instances of the systems in the families.

4.3 homotopy methods and path tracking algorithms

A homotopy method constructs a start system for an artificial parameter homotopy. Solution paths start at the known solutions of the start system and end at the solutions of the system that has to be solved, called the target system.

Path tracking algorithms compute approximations for the points on the solution paths, using predictor-corrector methods. The step size control can be done either *aposteriori*, based on the performance of the corrector; or *apriori*, using a predictor which estimates the distance to the nearest path and the convergence radius of the power series expansions of the solution curves. For natural parameter homotopies, arc length parameter continuation with aposteriori step size control is available.

4.3.1 functions in the module homotopies

A polynomial homotopy is a family of polynomial systems with one parameter. In an artificial parameter homotopy, there is a start and a target system. There is only one system in a natural parameter homotopy, where one variable plays the role of the parameter in the homotopy. The module homotopies exports several functions to set start and target functions in a homotopy. There are 36 copy functions:

copy_{double, double_double, quad_double}
_{laurent_system, system, solutions} _{from, into}_{start, target}

which take no input arguments (other than the verbose level), and have no return arguments (other than the return value of the call). Those copy functions are auxiliary to the set functions to define the homotopies.

homotopies.**clear_double_double_homotopy**(*vrblvl=0*)

Clears the homotopy set in double double precision.

homotopies.**clear_double_double_laurent_homotopy**(*vrblvl=0*)

Clears the Laurent homotopy set in double double precision.

homotopies.**clear_double_homotopy**(*vrblvl=0*)

Clears the homotopy set in double precision.

homotopies.**clear_double_laurent_homotopy**(*vrblvl=0*)

Clears the homotopy set in double precision.

homotopies.**clear_quad_double_homotopy**(*vrblvl=0*)

Clears the homotopy set in quad double precision.

homotopies.**clear_quad_double_laurent_homotopy**(*vrblvl=0*)

Clears the Laurent homotopy set in quad double precision.

homotopies.**copy_double_double_laurent_system_from_start**(*vrblvl=0*)

Copies the start Laurent system set in an artificial-parameter homotopy in double double precision into the Laurent system in double double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_double_laurent_system_from_target**(*vrblvl=0*)

Copies the target Laurent system set in an artificial-parameter homotopy in double double precision into the Laurent system in double double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_double_laurent_system_into_start**(*vrblvl=0*)

Copies the Laurent system set in double double precision to the start in an artificial-parameter homotopy in double double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_double_laurent_system_into_target**(*vrblvl=0*)

Copies the system set in double double precision to the target in an artificial-parameter homotopy in double double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_double_solutions_from_start**(*vrblvl=0*)

Copies the start solutions in an artificial-parameter homotopy in double double precision to the solutions in double double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_double_solutions_from_target**(*vrblvl=0*)

Copies the target solutions in an artificial-parameter homotopy in double precision to the solutions in double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_double_solutions_into_start**(*vrblvl=0*)

Copies the solutions set in double double precision to the start solutions in an artificial-parameter homotopy in double double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_double_solutions_into_target**(*vrblvl=0*)

Copies the solutions set in double double precision to the target solutions in an artificial-parameter homotopy in double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_double_system_from_start**(*vrblvl=0*)

Copies the start system set in an artificial-parameter homotopy in double double precision into the system in double double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_double_system_from_target**(*vrblvl=0*)

Copies the target system set in an artificial-parameter homotopy in double double precision into the system in double double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_double_system_into_start**(*vrblvl=0*)

Copies the system set in double double precision to the start in an artificial-parameter homotopy in double double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_double_system_into_target**(*vrblvl=0*)

Copies the system set in double double precision to the target in an artificial-parameter homotopy in double double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_laurent_system_from_start**(*vrblvl=0*)

Copies the start Laurent system set in an artificial-parameter homotopy in double precision into the Laurent system in double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_laurent_system_from_target**(*vrblvl=0*)

Copies the target Laurent system set in an artificial-parameter homotopy in double precision into the Laurent system in double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_laurent_system_into_start**(*vrblvl=0*)

Copies the Laurent system set in double precision to the start system in an artificial-parameter homotopy in double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_laurent_system_into_target**(*vrblvl=0*)

Copies the Laurent system set in double precision to the target in an artificial-parameter homotopy in double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_solutions_from_start**(*vrblvl=0*)

Copies the start solutions in an artificial-parameter homotopy in double precision to the solutions in double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_solutions_from_target**(*vrblvl=0*)

Copies the target solutions in an artificial-parameter homotopy in double precision to the solutions in double precision.

homotopies.**copy_double_solutions_into_start**(*vrblvl=0*)

Copies the solutions set in double precision to the start solutions in an artificial-parameter homotopy in double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_solutions_into_target**(*vrblvl=0*)

Copies the solutions set in double precision to the target solutions in an artificial-parameter homotopy in double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_system_from_start**(*vrblvl=0*)

Copies the start system set in an artificial-parameter homotopy in double precision into the system in double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_system_from_target**(*vrblvl=0*)

Copies the target system set in an artificial-parameter homotopy in double precision into the system in double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_system_into_start**(*vrblvl=0*)

Copies the system set in double precision to the start system in an artificial-parameter homotopy in double precision. The verbose level is given by vrblvl.

homotopies.**copy_double_system_into_target**(*vrblvl=0*)

Copies the system set in double precision to the target in an artificial-parameter homotopy in double precision. The verbose level is given by vrblvl.

homotopies.**copy_quad_double_laurent_system_from_start**(*vrblvl=0*)

Copies the start Laurent system set in an artificial-parameter homotopy in quad double precision into the Laurent system in quad double precision. The verbose level is given by vrblvl.

homotopies.**copy_quad_double_laurent_system_from_target**(*vrblvl=0*)

Copies the target Laurent system set in an artificial-parameter homotopy in quad double precision into the Laurent system in quad double precision. The verbose level is given by vrblvl.

homotopies.**copy_quad_double_laurent_system_into_start**(*vrblvl=0*)

Copies the Laurent system set in quad double precision to the start in an artificial-parameter homotopy in quad double precision. The verbose level is given by vrblvl.

homotopies.**copy_quad_double_laurent_system_into_target**(*vrblvl=0*)

Copies the Laurent system set in quad double precision to the target system in an artificial-parameter homotopy in quad double precision. The verbose level is given by vrblvl.

homotopies.**copy_quad_double_solutions_from_start**(*vrblvl=0*)

Copies the start solutions in an artificial-parameter homotopy in quad double precision to the solutions in quad double precision. The verbose level is given by vrblvl.

homotopies.**copy_quad_double_solutions_from_target**(*vrblvl=0*)

Copies the target solutions in an artificial-parameter homotopy in quad double precision to the solutions in quad double precision. The verbose level is given by vrblvl.

homotopies.**copy_quad_double_solutions_into_start**(*vrblvl=0*)

Copies the solutions set in quad double precision to the start solutions in an artificial-parameter homotopy in quad double precision. The verbose level is given by vrblvl.

homotopies.**copy_quad_double_solutions_into_target**(*vrblvl=0*)

Copies the solutions set in quad double precision to the target solutions in an artificial-parameter homotopy in double precision. The verbose level is given by vrblvl.

homotopies.**copy_quad_double_system_from_start**(*vrblvl=0*)

Copies the start system set in an artificial-parameter homotopy in quad double precision into the system in quad double precision. The verbose level is given by vrblvl.

homotopies.**copy_quad_double_system_from_target**(*vrblvl=0*)

Copies the target system set in an artificial-parameter homotopy in quad double precision into the system in quad double precision. The verbose level is given by vrblvl.

homotopies.**copy_quad_double_system_into_start**(*vrblvl=0*)

Copies the system set in quad double precision to the start in an artificial-parameter homotopy in quad double precision. The verbose level is given by vrblvl.

homotopies.**copy_quad_double_system_into_target**(*vrblvl=0*)

Copies the system set in quad double precision to the target system in an artificial-parameter homotopy in quad double precision. The verbose level is given by vrblvl.

homotopies.**example_start_system**(*vrblvl=0*)

Returns an example of a start system.

homotopies.**example_target_system**(*vrblvl=0*)

Returns an example of a target system.

homotopies.**get_double_double_target_solutions**(*vrblvl=0*)

Returns the list of target solutions computed in double double precision.

homotopies.**get_double_target_solutions**(*vrblvl=0*)

Returns the list of target solutions computed in double precision.

homotopies.**get_quad_double_target_solutions**(*vrblvl=0*)

Returns the list of target solutions computed in quad double precision.

homotopies.**main**()

Runs some tests.

homotopies.**set_double_double_homotopy**(*gamma=0*, *pwt=2*, *vrblvl=0*)

After the target and start system are set in double double precision, the homotopy is constructed with either a random gamma constant, or with the given complex value of gamma. The power of the continuation parameter is given by pwt. The gamma used to make the homotopy is returned.

homotopies.**set_double_double_laurent_homotopy**(*gamma=0*, *pwt=2*, *vrblvl=0*)

After the target and start Laurent system are set in double double precision, the homotopy is constructed with either a random gamma constant, or with the given complex value of gamma. The power of the continuation parameter is given by pwt. The gamma used to make the Laurent homotopy is returned.

homotopies.**set_double_double_laurent_start_system**(*pols*, *vrblvl=0*)

Sets the start Laurent system in an artificial parameter homotopy in double double precision to the list of polynomials in pols, which is assumed to be square. The verbose level is given by vrblvl.

homotopies.**set_double_double_laurent_target_system**(*pols*, *vrblvl=0*)

Sets the target Laurent system in an artificial parameter homotopy in double double precision to the list of polynomials in pols, which is assumed to be square. The verbose level is given by vrblvl.

homotopies.**set_double_double_start_solutions**(*nvr*, *sols*, *vrblvl=0*)

Sets the start solutions in an artificial parameter homotopy in double double precision to the list of solutions in sols, where the number of variables in nvr must match the dimension of the start system. The verbose level is given by vrblvl.

homotopies.**set_double_double_start_system**(*pols*, *vrblvl=0*)

Sets the start system in an artificial parameter homotopy in double double precision to the list of polynomials in pols, which is assumed to be square. The verbose level is given by vrblvl.

homotopies.**set_double_double_target_solutions**(*nvr*, *sols*, *vrblvl=0*)

Sets the target solutions in an artificial parameter homotopy in double double precision to the list of solutions in sols, where the number of variables in nvr must match the dimension of the start system. The verbose level is given by vrblvl.

homotopies.**set_double_double_target_system**(*pols*, *vrblvl=0*)

Sets the target system in an artificial parameter homotopy in double double precision to the list of polynomials in pols, which is assumed to be square. The verbose level is given by vrblvl.

homotopies.**set_double_homotopy**(*gamma=0*, *pwt=2*, *vrblvl=0*)

After the target and start system are set in double precision, the homotopy is constructed with either a random gamma constant, or with the given complex value of gamma. The power of the continuation parameter is given by pwt. The gamma used to make the homotopy is returned.

homotopies.**set_double_laurent_homotopy**(*gamma=0*, *pwt=2*, *vrblvl=0*)

After the target and start Laurent system are set in double precision, the homotopy is constructed with either a random gamma constant, or with the given complex value of gamma. The power of the continuation parameter is given by pwt. The gamma used to make the Laurent homotopy is returned.

homotopies.**set_double_laurent_start_system**(*pols*, *vrblvl=0*)

Sets the start Laurent system in an artificial parameter homotopy in double precision to the list of polynomials in pols, which is assumed to be square. The verbose level is given by vrblvl.

homotopies.**set_double_laurent_target_system**(*pols*, *vrblvl=0*)

Sets the target Laurent system in an artificial parameter homotopy in double precision to the list of polynomials in pols, which is assumed to be square. The verbose level is given by vrblvl.

homotopies.**set_double_start_solutions**(*nvr*, *sols*, *vrblvl=0*)

Sets the start solutions in an artificial parameter homotopy in double precision to the list of solutions in sols, where the number of variables in nvr must match the dimension of the start system. The verbose level is given by vrblvl.

homotopies.**set_double_start_system**(*pols*, *vrblvl=0*)

Sets the start system in an artificial parameter homotopy in double precision to the list of polynomials in pols, which is assumed to be square. The verbose level is given by vrblvl.

homotopies.**set_double_target_solutions**(*nvr*, *sols*, *vrblvl=0*)

Sets the target solutions in an artificial parameter homotopy in double precision to the list of solutions in sols, where the number of variables in nvr must match the dimension of the start system. The verbose level is given by vrblvl.

homotopies.**set_double_target_system**(*pols*, *vrblvl=0*)

Sets the target system in an artificial parameter homotopy in double precision to the list of polynomials in pols, which is assumed to be square. The verbose level is given by vrblvl.

homotopies.**set_quad_double_homotopy**(*gamma=0*, *pwt=2*, *vrblvl=0*)

After the target and start system are set in quad double precision, the homotopy is constructed with either a random gamma constant, or with the given complex value of gamma. The power of the continuation parameter is given by pwt. The gamma used to make the homotopy is returned.

homotopies.**set_quad_double_laurent_homotopy**(*gamma=0*, *pwt=2*, *vrblvl=0*)

After the target and start Laurent system are set in quad double precision, the homotopy is constructed with either a random gamma constant, or with the given complex value of gamma. The power of the continuation parameter is given by pwt. The gamma used to make the Laurent homotopy is returned.

homotopies.**set_quad_double_laurent_start_system**(*pols*, *vrblvl=0*)

Sets the start Laurent system in an artificial parameter homotopy in quad double precision to the list of polynomials in pols, which is assumed to be square. The verbose level is given by vrblvl.

homotopies.**set_quad_double_laurent_target_system**(*pols*, *vrblvl=0*)

Sets the target Laurent system in an artificial parameter homotopy in quad double precision to the list of polynomials in pols, which is assumed to be square. The verbose level is given by vrblvl.

homotopies.**set_quad_double_start_solutions**(*nvr*, *sols*, *vrblvl=0*)

Sets the start solutions in an artificial parameter homotopy in quad double precision to the list of solutions in sols, where the number of variables in nvr must match the dimension of the start system. The verbose level is given by vrblvl.

homotopies.**set_quad_double_start_system**(*pols*, *vrblvl=0*)

Sets the start system in an artificial parameter homotopy in quad double precision to the list of polynomials in pols, which is assumed to be square. The verbose level is given by vrblvl.

homotopies.**set_quad_double_target_solutions**(*nvr*, *sols*, *vrblvl=0*)

Sets the target solutions in an artificial parameter homotopy in quad double precision to the list of solutions in sols, where the number of variables in nvr must match the dimension of the start system. The verbose level is given by vrblvl.

homotopies.**set_quad_double_target_system**(*pols*, *vrblvl=0*)

Sets the target system in an artificial parameter homotopy in quad double precision to the list of polynomials in pols, which is assumed to be square. The verbose level is given by vrblvl.

homotopies.**test_double_double_laurent_start_system**(*vrblvl=0*)

Tests the definition of a start system in double double precision, set as Laurent system.

homotopies.**test_double_double_laurent_target_system**(*vrblvl=0*)

Tests the definition of a target system in double double precision, set as Laurent system.

homotopies.**test_double_double_start_system**(*vrblvl=0*)

Tests the definition of a start system in double double precision.

homotopies.**test_double_double_target_system**(*vrblvl=0*)

Tests the definition of a target system in double double precision.

homotopies.**test_double_laurent_start_system**(*vrblvl=0*)

Tests the definition of a start system in double precision, set as a Laurent system.

homotopies.**test_double_laurent_target_system**(*vrblvl=0*)

Tests the definition of a target system in double precision, set as Laurent system.

homotopies.**test_double_start_system**(*vrblvl=0*)

Tests the definition of a start system in double precision.

homotopies.**test_double_target_system**(*vrblvl=0*)

Tests the definition of a target system in double precision.

homotopies.**test_quad_double_laurent_start_system**(*vrblvl=0*)

Tests the definition of a start system in quad double precision, set as Laurent system.

homotopies.**test_quad_double_laurent_target_system**(*vrblvl=0*)

Tests the definition of a target system in quad double precision, set as Laurent system.

homotopies.**test_quad_double_start_system**(*vrblvl=0*)

Tests the definition of a start system in quad double precision.

homotopies.**test_quad_double_target_system**(*vrblvl=0*)

Tests the definition of a target system in quad double precision.

4.3.2 functions in the module starters

A an artificial parameter homotopy is a family of polynomial systems which connects a given target system to a start system. The module starters exports several functions to construct start systems. Analoguous to the culinary starters, start systems may be viewed as systems that are easier to solve (or digest) than general systems.

starters.**linear_product_root_count**(*pols*, *checkin=True*, *vrblvl=0*)

Given in *pols* a list of string representations of polynomials, returns a linear-product root count based on a supporting set structure of the polynomials in *pols*. This root count is an upper bound for the number of isolated solutions.

starters.**m_homogeneous_bezout_number**(*pols*, *vrblvl=0*)

Given in *pols* a list of string representations of polynomials, in as many variables as the elements in the list, this function applies a heuristic to generate a partition of the set of unknowns to exploit the product structure of the system. On return are the m-homogeneous Bezout number and the partition of the set of unknowns. If the partition equals the entire set of unknowns, then the 1-homogeneous Bezout number equals the total degree of the system.

starters.**m_homogeneous_start_system**(*pols*, *partition*, *checkin=True*, *vrblvl=0*)

For an m-homogeneous Bezout number of a polynomial system defined by a *partition* of the set of unknowns, one can define a linear-product system that has exactly as many regular solutions as the Bezount number. This linear-product system can then be used as start system in a homotopy to compute all isolated solutions of any polynomial system with the same m-homogeneous structure. This function returns a linear-product start system with random coefficients and its solutions for the given polynomials in pols and the partition. If *checkin*, then the list *pols* is tested to see if *pols* defines a square polynomial system. If the input system is not square, then an error message is printed and None is returned.

starters.**m_partition_bezout_number**(*pols*, *partition*, *vrblvl=0*)

There are as many m-homogeneous Bezout numbers as there are partitions of the set of unknowns of a polynomial system. Given in *pols* the string representations of a polynomial system in as many variables as equations, and a string representation of a *partition* of the set of unknowns, this function returns the m-homogeneous Bezout number corresponding to the given partition.

starters.**main**()

Runs some tests.

starters.**random_linear_product_system**(*pols*, *checkin=True*, *tosolve=True*, *vrblvl=0*)

Given in *pols* a list of string representations of polynomials, returns a random linear-product system based on a supporting set structure and its solutions as well (if *tosolve*). If *checkin*, then the list *pols* is tested to see if *pols* defines a square polynomial system. If the input system is not square, then an error message is printed and None is returned.

starters.**test_linear_product_root_count**(*vrblvl=0*)

Tests the linear product root count.

starters.**test_m_homogeneous_degree**(*vrblvl=0*)

Tests m-homogeneous Bezout number.

starters.**test_total_degree**(*vrblvl=0*)

Tests the total degree and the start system.

starters.**total_degree**(*pols*, *vrblvl=0*)

Given in pols a list of string representations of polynomials, returns the product of the degrees of the polynomials, the so-called total degree which bounds the number of isolated solutions of the polynomial system. The system is assumed to be square. The value of the verbose level is given by vrblvl.

starters.**total_degree_start_system**(*pols*, *checkin=True*, *vrblvl=0*)

Returns the system and solutions of the total degree start system for the polynomials represented by the strings in the list *pols*. If *checkin*, then the list *pols* is tested to see if *pols* defines a square polynomial system. If the input system is not square, then an error message is printed and None is returned.

4.3.3 functions in the module trackers

The module trackers offers functions to track paths starting at the known solutions of a start system and leading to the desired solutions of a target system, with aposteriori step size control algorithms. An aposteriori step size control algorithm determines the step size based on the performance of the corrector. For small problems, the default values of the parameters and tolerances for predictor and corrector suffice, otherwise they must be tuned. Reruns of paths must happen with the same value of the gamma constant.

trackers.**autotune_parameters**(*difficulty_level*, *digits_of_precision*, *vrblvl=0*)

Tunes the parameters given the difficulty level of the homotopy and the digits of working precision. The verbose level is given by vrblvl.

trackers.**clear_double_double_laurent_track_data**(*vrblvl=0*)

Clears the data allocated for path tracking in double double precision with an artificial parameter Laurent homotopy.

trackers.**clear_double_double_track_data**(*vrblvl=0*)

Clears the data allocated for path tracking in double double precision with an artificial parameter homotopy.

trackers.**clear_double_laurent_track_data**(*vrblvl=0*)

Clears the data allocated for path tracking in double precision with an artificial parameter Laurent homotopy.

trackers.**clear_double_track_data**(*vrblvl=0*)

Clears the data allocated for path tracking in double precision with an artificial parameter homotopy.

trackers.**clear_quad_double_laurent_track_data**(*vrblvl=0*)

Clears the data allocated for path tracking in quad double precision with an artificial parameter Laurent homotopy.

trackers.**clear_quad_double_track_data**(*vrblvl=0*)

Clears the data allocated for path tracking in quad double precision with an artificial parameter homotopy.

trackers.**do_double_double_laurent_track**(*tasks=0*, *vrblvl=0*)

Calls the path trackers in double double precision with a number of tasks equal to tasks (no multithreading if zero), for a homotopy of Laurent polynomial sysetms. The verbose level is given by vrblvl.

trackers.**do_double_double_track**(*tasks=0*, *vrblvl=0*)

Calls the path trackers in double double precision with a number of tasks equal to tasks (no multithreading if zero). The verbose level is given by vrblvl.

trackers.**do_double_laurent_track**(*tasks=0*, *vrblvl=0*)

Calls the path trackers in double precision with a number of tasks equal to tasks (no multithreading if zero), for a homotopy of Laurent polynomial systems. The verbose level is given by vrblvl.

trackers.**do_double_track**(*tasks=0*, *vrblvl=0*)

Calls the path trackers in double precision with a number of tasks equal to tasks (no multithreading if zero). The verbose level is given by vrblvl.

trackers.**do_quad_double_laurent_track**(*tasks=0*, *vrblvl=0*)

Calls the path trackers in quad double precision with a number of tasks equal to tasks (no multithreading if zero), for a homotopy of Laurent polynomial systems. The verbose level is given by vrblvl.

trackers.**do_quad_double_track**(*tasks=0*, *vrblvl=0*)

Calls the path trackers in quad double precision with a number of tasks equal to tasks (no multithreading if zero). The verbose level is given by vrblvl.

trackers.**double_double_laurent_track**(*target*, *start*, *startsols*, *gamma=0*, *pwt=2*, *tasks=0*, *vrblvl=0*)

Tracks paths in double double precision, for Laurent systems. On input are a target system, a start system with solutions, optionally: a (random) gamma constant and the number of tasks. The *target* is a list of strings representing the polynomials of the target system (which has to be solved). The *start* is a list of strings representing the polynomials of the start system, with known solutions in sols. The *startsols* is a list of strings representing start solutions. By default, a random *gamma* constant is generated, otherwise *gamma* must be a nonzero complex constant. The *pwt* is the power of t in the homotopy. Changing the default of *pwt* can only be done if a nonzero complex value for *gamma* is provided as well. The number of tasks in the multithreading is defined by *tasks*. The default zero value for *tasks* indicates no multithreading. On return is a tuple, with first the gamma used in the homotopy and then second, the string representations of the solutions computed at the end of the paths. Note: tasks=0 does not work …

trackers.**double_double_track**(*target*, *start*, *startsols*, *gamma=0*, *pwt=2*, *tasks=0*, *vrblvl=0*)

Tracks paths in double double precision. On input are a target system, a start system with solutions, optionally: a (random) gamma constant and the number of tasks. The *target* is a list of strings representing the polynomials of the target system (which has to be solved). The *start* is a list of strings representing the polynomials of the start system, with known solutions in sols. The *startsols* is a list of strings representing start solutions. By default, a random *gamma* constant is generated, otherwise *gamma* must be a nonzero complex constant. The *pwt* is the power of t in the homotopy. Changing the default of *pwt* can only be done if a nonzero complex value for *gamma* is provided as well. The number of tasks in the multithreading is defined by *tasks*. The default zero value for *tasks* indicates no multithreading. On return is a tuple, with first the gamma used in the homotopy and then second, the string representations of the solutions computed at the end of the paths. Note: tasks=0 does not work ...

trackers.**double_laurent_track**(*target*, *start*, *startsols*, *gamma=0*, *pwt=2*, *tasks=0*, *vrblvl=0*)

Track paths in double precision, for Laurent systems. On input are a target system, a start system with solutions, optionally: a (random) gamma constant and the number of tasks. The *target* is a list of strings representing the polynomials of the target system (which has to be solved). The *start* is a list of strings representing the polynomials of the start system, with known solutions in sols. The *startsols* is a list of strings representing start solutions. By default, a random *gamma* constant is generated, otherwise *gamma* must be a nonzero complex constant. The *pwt* is the power of t in the homotopy. Changing the default of *pwt* can only be done if a nonzero complex value for *gamma* is provided as well. The number of tasks in the multithreading is defined by *tasks*. The default zero value for *tasks* indicates no multithreading. On return is a tuple, with first the gamma used in the homotopy and then second, the string representations of the solutions computed at the end of the paths.

trackers.**double_track**(*target*, *start*, *startsols*, *gamma=0*, *pwt=2*, *tasks=0*, *vrblvl=0*)

Track paths in double precision. On input are a target system, a start system with solutions, optionally: a (random) gamma constant and the number of tasks. The *target* is a list of strings representing the polynomials of the target system (which has to be solved). The *start* is a list of strings representing the polynomials of the start system, with known solutions in sols. The *startsols* is a list of strings representing start solutions. By default, a random *gamma* constant is generated, otherwise *gamma* must be a nonzero complex constant. The *pwt* is the power of t in the homotopy. Changing the default of *pwt* can only be done if a nonzero complex value for *gamma* is provided as well. The number of tasks in the multithreading is defined by *tasks*. The default zero value for *tasks* indicates no multithreading. On return is a tuple, with first the gamma used in the homotopy and then second, the string representations of the solutions computed at the end of the paths.

trackers.**get_condition_level**(*vrblvl=0*)

Returns the level of difficulty. The verbose level is given by vrblvl.

trackers.**get_parameter_value**(*idx*, *vrblvl=0*)

Returns the value of the parameter with index idx. The verbose level is given by vrblvl.

trackers.**initialize_double_double_solution**(*nvr*, *sol*, *vrblvl=0*)

A double double precision path tracker with a generator is initialized with a start solution *sol* in a number of variables equal to the value of *nvr*.

trackers.**initialize_double_double_tracker**(*target*, *start*, *fixedgamma=True*, *regamma=0.0*, *imgamma=0.0*, *vrblvl=0*)

Initializes a path tracker with a generator for a *target* and *start* system given in double double precision. If *fixedgamma*, then gamma will be a fixed default value, otherwise, a random complex constant for gamma is generated, but only if *regamma* and *imgamma* are both equal to 0.0. If not *fixedgamma* and moreover: *regamma* and *imgamma* are not both zero, then the complex number with real part in *regamma* and imaginary part in *imgamma* will be the gamma constant.

trackers.**initialize_double_solution**(*nvr*, *sol*, *vrblvl=0*)

A double precision path tracker with a generator is initialized with a start solution *sol* in a number of variables equal to the value of *nvr*.

trackers.**initialize_double_tracker**(*target*, *start*, *fixedgamma=True*, *regamma=0.0*, *imgamma=0.0*, *vrblvl=0*)

Initializes a path tracker with a generator for a *target* and *start* system given in standard double precision. If *fixedgamma*, then gamma will be a fixed default value, otherwise, a random complex constant for gamma is generated, but only if *regamma* and *imgamma* are both equal to 0.0. If not *fixedgamma* and moreover: *regamma* and *imgamma* are not both zero, then the complex number with real part in *regamma* and imaginary part in *imgamma* will be the gamma constant.

trackers.**initialize_quad_double_solution**(*nvr*, *sol*, *vrblvl=0*)

A quad double precision path tracker with a generator is initialized with a start solution *sol* in a number of variables equal to the value of *nvr*.

trackers.**initialize_quad_double_tracker**(*target*, *start*, *fixedgamma=True*, *regamma=0.0*, *imgamma=0.0*, *vrblvl=0*)

Initializes a path tracker with a generator for a *target* and *start* system given in quad double precision. If *fixedgamma*, then gamma will be a fixed default value, otherwise, a random complex constant for gamma is generated, but only if *regamma* and *imgamma* are both equal to 0.0. If not *fixedgamma* and moreover: *regamma* and *imgamma* are not both zero, then the complex number with real part in *regamma* and imaginary part in *imgamma* will be the gamma constant.

trackers.**interactive_tune**(*vrblvl=0*)

Interactive tuning of the parameters. The verbose level is given by vrblvl.

trackers.**main**()

Runs some tests on tuning and tracking.

trackers.**next_double_double_solution**(*vrblvl=0*)

Returns the next solution on a path tracked with double double precision arithmetic, provided the functions **initialize_double_double_tracker()** and **initialize_double_double_solution()** have been executed properly.

trackers.**next_double_solution**(*vrblvl=0*)

Returns the next solution on a path tracked with double precision arithmetic, provided the functions **initialize_double_tracker()** and **initialize_double_solution()** have been executed properly.

trackers.**next_quad_double_solution**(*vrblvl=0*)

Returns the next solution on a path tracked with quad double precision arithmetic, provided the functions **initialize_quad_double_tracker()** and **initialize_quad_double_solution()** have been executed properly.

trackers.**quad_double_laurent_track**(*target*, *start*, *startsols*, *gamma=0*, *pwt=2*, *tasks=0*, *vrblvl=0*)

Tracks paths in quad double precision, for Laurent systems. On input are a target system, a start system with solutions, optionally: a (random) gamma constant and the number of tasks. The *target* is a list of strings representing the polynomials of the target system (which has to be solved). The *start* is a list of strings representing the polynomials of the start system, with known solutions in sols. The *startsols* is a list of strings representing start solutions. By default, a random *gamma* constant is generated, otherwise *gamma* must be a nonzero complex constant. The *pwt* is the power of t in the homotopy. Changing the default of *pwt* can only be done if a nonzero complex value for *gamma* is provided as well. The number of tasks in the multithreading is defined by *tasks*. The default zero value for *tasks* indicates no multithreading. On return is a tuple, with first the gamma used in the homotopy and then second, the string representations of the solutions computed at the end of the paths. Note: tasks=0 does not work ...

trackers.**quad_double_track**(*target*, *start*, *startsols*, *gamma=0*, *pwt=2*, *tasks=0*, *vrblvl=0*)

Tracks paths in quad double precision. On input are a target system, a start system with solutions, optionally: a (random) gamma constant and the number of tasks. The *target* is a list of strings representing the polynomials of the target system (which has to be solved). The *start* is a list of strings representing the polynomials of the start system, with known solutions in sols. The *startsols* is a list of strings representing start solutions. By default, a random *gamma* constant is generated, otherwise *gamma* must be a nonzero complex constant. The *pwt* is the

power of t in the homotopy. Changing the default of *pwt* can only be done if a nonzero complex value for *gamma* is provided as well. The number of tasks in the multithreading is defined by *tasks*. The default zero value for *tasks* indicates no multithreading. On return is a tuple, with first the gamma used in the homotopy and then second, the string representations of the solutions computed at the end of the paths. Note: tasks=0 does not work ...

trackers.**set_condition_level**(*level*, *vrblvl=0*)

Sets the parameter that represents the difficulty level of the homotopy to the value of *level*. The default level equals zero, higher values lead to smaller tolerances. On return is the failure code, which is zero if all went well.

trackers.**set_parameter_value**(*idx*, *value*, *vrblvl=0*)

Sets the parameter with index idx to the given value. The verbose level is given by vrblvl.

trackers.**show_parameters**(*vrblvl=0*)

Displays the current values of the continuation parameters. The verbose level is given by vrblvl.

trackers.**test_double_double_laurent_track**(*vrblvl=0*)

Tests tracking the mickey mouse example of two quadrics, set as Laurent system in double double precision.

trackers.**test_double_double_track**(*vrblvl=0*)

Tests tracking the mickey mouse example of two quadrics, in double double precision.

trackers.**test_double_laurent_track**(*vrblvl=0*)

Tests tracking the mickey mouse example of two quadrics, set as Laurent system in double precision.

trackers.**test_double_track**(*vrblvl=0*)

Tests tracking the mickey mouse example of two quadrics, in double precision.

trackers.**test_next_double_double_track**(*vrblvl=0*)

Tests the step-by-step tracking on the mickey mouse example of two quadrics, in double double precision.

trackers.**test_next_double_track**(*vrblvl=0*)

Tests the step-by-step tracking on the mickey mouse example of two quadrics, in double precision.

trackers.**test_next_quad_double_track**(*vrblvl=0*)

Tests the step-by-step tracking on the mickey mouse example of two quadrics, in quad double precision.

trackers.**test_quad_double_laurent_track**(*vrblvl=0*)

Tests tracking the mickey mouse example of two quadrics, set as Laurent system in quad double precision.

trackers.**test_quad_double_track**(*vrblvl=0*)

Tests tracking the mickey mouse example of two quadrics, in quad double precision.

trackers.**test_tuning**(*vrblvl=0*)

Runs some tests on tuning the parameters.

trackers.**test_write_parameters**(*vrblvl=0*)

Tests the writing of the parameters.

trackers.**write_parameters**(*vrblvl=0*)

Writes the parameters with repeated calls to get_parameter_value(), as the show_parameters() does not work in a Jupyter notebook.

4.3.4 functions in the module tropisms

The module tropisms exports functions to manage numerically computed tropisms in double, double double, or quad double precision, in a polyhedral end game with aposteriori step size control.

tropisms.**clear_double_double_tropisms**(*vrblvl=0*)

Clears the tropisms in double double precision.

tropisms.**clear_double_tropisms**(*vrblvl=0*)

Clears the tropisms in double precision.

tropisms.**clear_quad_double_tropisms**(*vrblvl=0*)

Clears the tropisms in quad double precision.

tropisms.**double_double_initialize_tropisms**(*nbt*, *dim*, *wnd*, *dirs*, *errs*, *vrblvl=0*)

Initializes the tropisms, given in double double precision, along with estimates for their winding numbers and errors. On entry are the following five parameters:

nbt: the number of direction vectors;

dim: the number of coordinates in each vector;

wnd: a list of integer values for the winding numbers, as many as *nbt*;

dirs: a list of lists of doubles with the coordinates of the directions, each inner list has *dim* doubles and *nbt* vectors are given;

errs: a list of *nbt* doubles.

tropisms.**double_double_tropisms_dimension**(*vrblvl=0*)

Returns the dimension of tropisms in double double precision.

tropisms.**double_double_tropisms_number**(*vrblvl=0*)

Returns the number of tropisms in double double precision.

tropisms.**double_initialize_tropisms**(*nbt*, *dim*, *wnd*, *dirs*, *errs*, *vrblvl=0*)

Initializes the tropisms, given in double precision, along with estimates for their winding numbers and errors. On entry are the following five parameters:

nbt: the number of direction vectors;

dim: the number of coordinates in each vector;

wnd: a list of integer values for the winding numbers, as many as *nbt*;

dirs: a list of lists of doubles with the coordinates of the directions, each inner list has *dim* doubles and *nbt* vectors are given;

errs: a list of *nbt* doubles.

tropisms.**double_tropisms_dimension**(*vrblvl=0*)

Returns the dimension of tropisms in double precision.

tropisms.**double_tropisms_number**(*vrblvl=0*)

Returns the number of tropisms in double precision.

tropisms.**get_double_double_tropisms**(*nbt*, *dim*, *vrblvl=0*)

Given on input the number of tropisms in *nbt* and the dimension in *dim*, returns a tuple of three lists: the winding numbers, coordinates of the direction vectors, and the errors; in double double precision.

tropisms.**get_double_tropisms**(*nbt*, *dim*, *vrblvl=0*)

Given on input the number of tropisms in *nbt* and the dimension in *dim*, returns a tuple of three lists: the winding numbers, coordinates of the direction vectors, and the errors; in double precision.

tropisms.**get_quad_double_tropisms**(*nbt*, *dim*, *vrblvl=0*)

Given on input the number of tropisms in *nbt* and the dimension in *dim*, returns a tuple of three lists: the winding numbers, coordinates of the direction vectors, and the errors; in quad double precision.

tropisms.**main**()

Runs some tests.

tropisms.**quad_double_initialize_tropisms**(*nbt*, *dim*, *wnd*, *dirs*, *errs*, *vrblvl=0*)

Initializes the tropisms, given in quad double precision, along with estimates for their winding numbers and errors. On entry are the following five parameters:

nbt: the number of direction vectors;

dim: the number of coordinates in each vector;

wnd: a list of integer values for the winding numbers, as many as *nbt*;

dirs: a list of lists of doubles with the coordinates of the directions, each inner list has *dim* doubles and *nbt* vectors are given;

errs: a list of *nbt* doubles.

tropisms.**quad_double_tropisms_dimension**(*vrblvl=0*)

Returns the dimension of tropisms in quad double precision.

tropisms.**quad_double_tropisms_number**(*vrblvl=0*)

Returns the number of tropisms in quad double precision.

tropisms.**test_double_double_endgame**(*vrblvl=0*)

Tests the numerical computation of a tropism, in double double precision.

tropisms.**test_double_double_tropisms_data**(*vrblvl=0*)

Tests tropisms data in double double precision.

tropisms.**test_double_endgame**(*vrblvl=0*)

Tests the numerical computation of a tropism, in double precision.

tropisms.**test_double_tropisms_data**(*vrblvl=0*)

Tests tropisms data in double precision.

tropisms.**test_quad_double_endgame**(*vrblvl=0*)

Tests the numerical computation of a tropism, in quad double precision.

tropisms.**test_quad_double_tropisms_data**(*vrblvl=0*)

Tests tropisms data in quad double precision.

4.3.5 functions in the module sweepers

The module sweepers exports the definition of sweep homotopies and the tracking of solution paths defined by sweep homotopies. A sweep homotopy is a polynomial system where some of the variables are considered as parameters. Given solutions for some parameters and new values for the parameters, we can track the solution paths starting at the given solutions and ending at the new solutions for the new values of the parameters. The sweep is controlled by a convex linear combination between the list of start and target values for the parameters. We distinguish between a complex and a real sweep. In a complex sweep, with a randomly generated gamma we avoid singularities along the solution paths, in a complex convex combination between the start and target values for the parameters. This complex sweep is applicable only when the parameter space is convex. In a real sweep, arc-length parameter continuation is applied. The algorithms applied in this module are described in the paper by Kathy Piret and Jan Verschelde: Sweeping Algebraic Curves for Singular Solutions. Journal of Computational and Applied Mathematics, volume 234, number 4, pages 1228-1237, 2010.

`sweepers.`**`double_complex_sweep`**(*pols*, *sols*, *nvar*, *pars*, *start*, *target*, *vrblvl=0*)

For the polynomials in the list of strings *pols* and the solutions in *sols* for the values in the list *start*, a sweep through the parameter space will be performed in double precision to the target values of the parameters in the list *target*. The number of variables in the polynomials and the solutions must be the same and be equal to the value of *nvar*. The list of symbols in *pars* contains the names of the variables in the polynomials *pols* that serve as parameters. The size of the lists *pars*, *start*, and *target* must be same.

`sweepers.`**`double_complex_sweep_run`**(*gchoice*, *regamma*, *imgamma*, *vrblvl=0*)

Starts the trackers in a complex convex parameter homotopy, in double precision, where the indices to the parameters, start and target values are already defined, and the systems and solution are set in double precision. The input parameter gchoice is 0, 1, or 2, for respectively a randomly generated gamma (0), or no gamma (1), or a user given gamma with real and imaginary parts in regamma and imgamma. With a random gamma, this is known as cheater's homotopy.

`sweepers.`**`double_double_complex_sweep`**(*pols*, *sols*, *nvar*, *pars*, *start*, *target*, *vrblvl=0*)

For the polynomials in the list of strings *pols* and the solutions in *sols* for the values in the list *start*, a sweep through the parameter space will be performed in double double precision to the target values of the parameters in the list *target*. The number of variables in the polynomials and the solutions must be the same and be equal to the value of *nvar*. The list of symbols in *pars* contains the names of the variables in the polynomials *pols* that serve as parameters. The size of the lists *pars*, *start*, and *target* must be same.

`sweepers.`**`double_double_complex_sweep_run`**(*gchoice*, *regamma*, *imgamma*, *vrblvl=0*)

Starts the trackers in a complex convex parameter homotopy, in double double precision, where the indices to the parameters, start and target values are already defined, and the systems and solution are set in double double precision. The input parameter gchoice is 0, 1, or 2, for respectively a randomly generated gamma (0), or no gamma (1), or a user given gamma with real and imaginary parts in regamma and imgamma. With a random gamma, this is known as cheater's homotopy.

`sweepers.`**`double_double_real_sweep`**(*pols*, *sols*, *par='s'*, *start=0.0*, *target=1.0*, *vrblvl=0*)

A real sweep homotopy is a family of n equations in n+1 variables, where one of the variables is the artificial parameter s which moves from 0.0 to 1.0. The last equation can then be of the form

(1 - s)*(lambda - L[0]) + s*(lambda - L[1]) = 0 so that,

at s = 0, the natural parameter lambda has the value L[0], and

at s = 1, the natural parameter lambda has the value L[1].

Thus: as s moves from 0 to 1, lambda goes from L[0] to L[1].

All solutions in the list *sols* must have then the value L[0] for the variable lambda. The sweep stops when the target value for s is reached or when a singular solution is encountered. Computations happen in double double precision.

sweepers.**double_double_real_sweep_run**(*vrblvl=0*)

Starts a sweep with a natural parameter in a family of n equations in n+1 variables, where the last variable is the artificial parameter s that moves the one natural parameter from a start to target value. The last equation is of the form (1-s)*(A - v[0]) + s*(A - v[1]), where A is the natural parameter, going from the start value v[0] to the target value v[1], already set in double double precision. Also set before calling the function are the homotopy and the start solutions, where every solution has the value v[0] for the A variable. The sweep stops when s reaches the value v[1], or when a singularity is encountered on the path.

sweepers.**double_real_sweep**(*pols*, *sols*, *par='s'*, *start=0.0*, *target=1.0*, *vrblvl=0*)

A real sweep homotopy is a family of n equations in n+1 variables, where one of the variables is the artificial parameter s which moves from 0.0 to 1.0. The last equation can then be of the form

(1 - s)*(lambda - L[0]) + s*(lambda - L[1]) = 0 so that,

at s = 0, the natural parameter lambda has the value L[0], and

at s = 1, the natural parameter lambda has the value L[1].

Thus: as s moves from 0 to 1, lambda goes from L[0] to L[1].

All solutions in the list *sols* must have then the value L[0] for the variable lambda. The sweep stops when the target value for s is reached or when a singular solution is encountered. Computations happen in double precision.

sweepers.**double_real_sweep_run**(*vrblvl=0*)

Starts a sweep with a natural parameter in a family of n equations in n+1 variables, where the last variable is the artificial parameter s that moves the one natural parameter from a start to target value. The last equation is of the form (1-s)*(A - v[0]) + s*(A - v[1]), where A is the natural parameter, going from the start value v[0] to the target value v[1], already set in double precision. Also set before calling the function are the homotopy and the start solutions, where every solution has the value v[0] for the A variable. The sweep stops when s reaches the value v[1], or when a singularity is encountered on the path.

sweepers.**main**()

Runs some tests.

sweepers.**quad_double_complex_sweep**(*pols*, *sols*, *nvar*, *pars*, *start*, *target*, *vrblvl=0*)

For the polynomials in the list of strings *pols* and the solutions in *sols* for the values in the list *start*, a sweep through the parameter space will be performed in double double precision to the target values of the parameters in the list *target*. The number of variables in the polynomials and the solutions must be the same and be equal to the value of *nvar*. The list of symbols in *pars* contains the names of the variables in the polynomials *pols* that serve as parameters. The size of the lists *pars*, *start*, and *target* must be same.

sweepers.**quad_double_complex_sweep_run**(*gchoice*, *regamma*, *imgamma*, *vrblvl=0*)

Starts the trackers in a complex convex parameter homotopy, in quad double precision, where the indices to the parameters, start and target values are already defined, and the systems and solution are set in double double precision. The input parameter gchoice is 0, 1, or 2, for respectively a randomly generated gamma (0), or no gamma (1), or a user given gamma with real and imaginary parts in regamma and imgamma. With a random gamma, this is known as cheater's homotopy.

sweepers.**quad_double_real_sweep**(*pols*, *sols*, *par='s'*, *start=0.0*, *target=1.0*, *vrblvl=0*)

A real sweep homotopy is a family of n equations in n+1 variables, where one of the variables is the artificial parameter s which moves from 0.0 to 1.0. The last equation can then be of the form

(1 - s)*(lambda - L[0]) + s*(lambda - L[1]) = 0 so that,

at s = 0, the natural parameter lambda has the value L[0], and

at s = 1, the natural parameter lambda has the value L[1].

Thus: as s moves from 0 to 1, lambda goes from L[0] to L[1].

All solutions in the list *sols* must have then the value L[0] for the variable lambda. The sweep stops when the target value for s is reached or when a singular solution is encountered. Computations happen in quad double precision.

sweepers.**quad_double_real_sweep_run**(*vrblvl=0*)

Starts a sweep with a natural parameter in a family of n equations in n+1 variables, where the last variable is the artificial parameter s that moves the one natural parameter from a start to target value. The last equation is of the form (1-s)*(A - v[0]) + s*(A - v[1]), where A is the natural parameter, going from the start value v[0] to the target value v[1], already set in quad double precision. Also set before calling the function are the homotopy and the start solutions, where every solution has the value v[0] for the A variable. The sweep stops when s reaches the value v[1], or when a singularity is encountered on the path.

sweepers.**set_double_double_start**(*start*, *vrblvl=0*)

Sets the values of all parameters to start, which contains the consecutive values of all real and imaginary parts of the start values of all parameters, in double double precision. The length of start must be twice the size of the start in double precision.

sweepers.**set_double_double_target**(*target*, *vrblvl=0*)

Sets the values of all parameters to target, which contains the consecutive values of all real and imaginary parts of the target values of all parameters, in double double precision. The length of start must be twice the size of the start in double precision.

sweepers.**set_double_start**(*start*, *vrblvl=0*)

Sets the values of all parameters to start, which contains the consecutive values of all real and imaginary parts of the start values of all parameters, in double precision.

sweepers.**set_double_target**(*target*, *vrblvl=0*)

Sets the values of all parameters to target, which contains the consecutive values of all real and imaginary parts of the target values of all parameters, in double precision.

sweepers.**set_parameter_names**(*neq*, *nvr*, *pars*, *vrblvl=0*)

Defines which variables serve as parameters, by providing the names of the parameters in the list pars. The number of equations is given in neq and the number of variables is in nvr.

sweepers.**set_quad_double_start**(*start*, *vrblvl=0*)

Sets the values of all parameters to start, which contains the consecutive values of all real and imaginary parts of the start values of all parameters, in quad double precision. The length of start must be four times the size of the start in double precision.

sweepers.**set_quad_double_target**(*target*, *vrblvl=0*)

Sets the values of all parameters to target, which contains the consecutive values of all real and imaginary parts of the target values of all parameters, in double double precision. The length of start must be four times the size of the start in double precision.

sweepers.**test_double_complex_sweep**(*vrblvl=0*)

Runs a complex sweep on two points on the unit circle. Although we start at two points with real coordinates and we end at two points that have nonzero imaginary parts, the sweep does not encounter a singularity because of the random complex gamma constant.

sweepers.**test_double_double_complex_sweep**(*vrblvl=0*)

Runs a complex sweep on two points on the unit circle, in double double precision. Although we start at two points with real coordinates and we end at two points that have nonzero imaginary parts, the sweep does not encounter a singularity because of the random complex gamma constant.

sweepers.**test_double_double_real_sweep**(*vrblvl=0*)

Runs a real sweep on two points on the unit circle: (1,0), (-1,0), moving the second coordinate from 0 to 2. The sweep will stop at the quadratic turning point: (0,1). We can also run the sweep starting at two complex points:

(2*j, sqrt(5)) and (-2*j, sqrt(5)), moving the second coordinate from sqrt(5) to 0. This sweep will also stop at (0,1).

sweepers.**test_double_real_sweep**(*vrblvl=0*)

Runs a real sweep on two points on the unit circle: (1,0), (-1,0), moving the second coordinate from 0 to 2. The sweep will stop at the quadratic turning point: (0,1). We can also run the sweep starting at two complex points: (2*j, sqrt(5)) and (-2*j, sqrt(5)), moving the second coordinate from sqrt(5) to 0. This sweep will also stop at (0,1).

sweepers.**test_quad_double_complex_sweep**(*vrblvl=0*)

Runs a complex sweep on two points on the unit circle, in quad double precision. Although we start at two points with real coordinates and we end at two points that have nonzero imaginary parts, the sweep does not encounter a singularity because of the random complex gamma constant.

sweepers.**test_quad_double_real_sweep**(*vrblvl=0*)

Runs a real sweep on two points on the unit circle: (1,0), (-1,0), moving the second coordinate from 0 to 2. The sweep will stop at the quadratic turning point: (0,1). We can also run the sweep starting at two complex points: (2*j, sqrt(5)) and (-2*j, sqrt(5)), moving the second coordinate from sqrt(5) to 0. This sweep will also stop at (0,1).

4.3.6 functions in the module series

Exports functions to compute series of solution curves defined by polynomial homotopies using Newton's method.

series.**checkin_newton_at_series**(*nbsym*, *lser*, *idx*)

Given in nbsym the number of symbols in the polynomial system, in lser the list of leading terms in the series and in idx the index of the parameter, returns True if nbsym = len(lser) if idx == 0, or otherwise if nbsym = len(lser) + 1 if idx != 0. An error message is written and False is returned if the above conditions are not satisfied.

series.**double_double_newton_at_point**(*pols*, *sols*, *idx=1*, *maxdeg=4*, *nbr=4*, *vrblvl=0*)

Computes series in double double precision for the polynomials in *pols*, where the leading coefficients are the solutions in *sols*. On entry are the following five parameters:

pols: a list of string representations of polynomials,

sols: a list of regular solutions of the polynomials in pols,

idx: index of the series parameter, by default equals 1,

maxdeg: maximal degree of the series,

nbr: number of steps with Newton's method,

vrblvl: the verbose level.

On return is a list of lists of strings. Each lists of strings represents the series solution for the variables in the list *pols*.

series.**double_double_newton_at_series**(*pols*, *lser*, *idx=1*, *maxdeg=4*, *nbr=4*, *checkin=True*, *vrblvl=0*)

Computes series in double double precision for the polynomials in *pols*, where the leading terms are given in the list *lser*. On entry are the following five parameters:

pols: a list of string representations of polynomials,

lser: a list of polynomials in the series parameter (e.g.: t), for use as start terms in Newton's method,

idx: index of the series parameter, by default equals 1,

maxdeg: maximal degree of the series,

nbr: number of steps with Newton's method,

checkin: checks whether the number of symbols in pols matches the length of the list lser if idx == 0, or is one less than the length of the list lser if idx != 0. If the conditions are not satisfied, then an error message is printed and lser is returned.

vrblvl: is the verbose level.

On return is a list of lists of strings. Each lists of strings represents the series solution for the variables in the list *pols*.

series.**double_double_pade_approximants**(*pols*, *sols*, *idx=1*, *numdeg=2*, *dendeg=2*, *nbr=4*, *vrblvl=0*)

Computes Pade approximants based on the series in double double precision for the polynomials in *pols*, where the leading coefficients of the series are the solutions in *sols*. On entry are the following seven parameters:

pols: a list of string representations of polynomials,

sols: a list of solutions of the polynomials in *pols*,

idx: index of the series parameter, by default equals 1,

numdeg: the degree of the numerator,

dendeg: the degree of the denominator,

nbr: number of steps with Newton's method,

vrblvl: is the verbose level.

On return is a list of lists of strings. Each lists of strings represents the series solution for the variables in the list *pols*.

series.**double_newton_at_point**(*pols*, *sols*, *idx=1*, *maxdeg=4*, *nbr=4*, *vrblvl=0*)

Computes series in double precision for the polynomials in *pols*, where the leading coefficients are the solutions in *sols*. On entry are the following five parameters:

pols: a list of string representations of polynomials,

sols: a list of regular solutions of the polynomials in pols,

idx: index of the series parameter, by default equals 1,

maxdeg: maximal degree of the series,

nbr: number of steps with Newton's method,

vrblvl: the verbose level.

On return is a list of lists of strings. Each lists of strings represents the series solution for the variables in the list *pols*.

series.**double_newton_at_series**(*pols*, *lser*, *idx=1*, *maxdeg=4*, *nbr=4*, *checkin=True*, *vrblvl=0*)

Computes series in double precision for the polynomials in *pols*, where the leading terms are given in the list *lser*. On entry are the following five parameters:

pols: a list of string representations of polynomials,

lser: a list of polynomials in the series parameter (e.g.: t), for use as start terms in Newton's method,

idx: index of the series parameter, by default equals 1,

maxdeg: maximal degree of the series,

nbr: number of steps with Newton's method,

checkin: checks whether the number of symbols in pols matches the length of the list lser if idx == 0, or is one less than the length of the list lser if idx != 0. If the conditions are not satisfied, then an error message is printed and lser is returned.

vrblvl: is the verbose level.

On return is a list of lists of strings. Each lists of strings represents the series solution for the variables in the list *pols*.

series.**double_pade_approximants**(*pols*, *sols*, *idx=1*, *numdeg=2*, *dendeg=2*, *nbr=4*, *vrblvl=0*)

Computes Pade approximants based on the series in double precision for the polynomials in *pols*, where the leading coefficients of the series are the solutions in *sols*. On entry are the following seven parameters:

pols: a list of string representations of polynomials,

sols: a list of solutions of the polynomials in *pols*,

idx: index of the series parameter, by default equals 1,

numdeg: the degree of the numerator,

dendeg: the degree of the denominator,

nbr: number of steps with Newton's method,

vrblvl: is the verbose level.

On return is a list of lists of strings. Each lists of strings represents the series solution for the variables in the list *pols*.

series.**main**()

Runs some tests on series developments.

series.**make_fractions**(*pols*)

Given a list of string representations for the numerator and denominator polynomials in its even and odd numbered indices, returns a list of string representations for the fractions.

series.**quad_double_newton_at_point**(*pols*, *sols*, *idx=1*, *maxdeg=4*, *nbr=4*, *vrblvl=0*)

Computes series in quad double precision for the polynomials in *pols*, where the leading coefficients are the solutions in *sols*. On entry are the following five parameters:

pols: a list of string representations of polynomials,

sols: a list of solutions of the polynomials in pols,

idx: index of the series parameter, by default equals 1,

maxdeg: maximal degree of the series,

nbr: number of steps with Newton's method,

vrblvl: the verbose level.

On return is a list of lists of strings. Each lists of strings represents the series solution for the variables in the list *pols*.

series.**quad_double_newton_at_series**(*pols*, *lser*, *idx=1*, *maxdeg=4*, *nbr=4*, *checkin=True*, *vrblvl=0*)

Computes series in quad double precision for the polynomials in *pols*, where the leading terms are given in the list *lser*. On entry are the following five parameters:

pols: a list of string representations of polynomials,

lser: a list of polynomials in the series parameter (e.g.: t), for use as start terms in Newton's method,

idx: index of the series parameter, by default equals 1,

maxdeg: maximal degree of the series,

nbr: number of steps with Newton's method,

checkin: checks whether the number of symbols in pols matches the length of the list lser if idx == 0, or is one less than the length of the list lser if idx != 0. If the conditions are not satisfied, then an error message is printed and lser is returned.

vrblvl: is the verbose level.

On return is a list of lists of strings. Each lists of strings represents the series solution for the variables in the list *pols*.

series.**quad_double_pade_approximants**(*pols*, *sols*, *idx=1*, *numdeg=2*, *dendeg=2*, *nbr=4*, *vrblvl=0*)

Computes Pade approximants based on the series in quad double precision for the polynomials in *pols*, where the leading coefficients of the series are the solutions in *sols*. On entry are the following seven parameters:

pols: a list of string representations of polynomials,

sols: a list of solutions of the polynomials in *pols*,

idx: index of the series parameter, by default equals 1,

numdeg: the degree of the numerator,

dendeg: the degree of the denominator,

nbr: number of steps with Newton's method,

vrblvl: is the verbose level.

On return is a list of lists of strings. Each lists of strings represents the series solution for the variables in the list *pols*.

series.**rational_forms**(*pols*)

Given a list of lists of string representations for the numerators and denominators, returns the proper rational representations for the Pade approximants.

series.**replace_symbol**(*pol*, *idx*, *vrblvl=0*)

In the polynomial pol, replaces the first symbol by the symbol at place idx.

series.**substitute_symbol**(*pols*, *idx*, *vrblvl=0*)

Given in pols is a list of polynomials, replaces the first symbol by the symbol at place idx.

series.**test_double_apollonius_at_series**(*vrblvl=0*)

Tests Newton's method starting at a series for the problem of Apollonius, in double precision. The parameter t is the fourth variable, whence we call Newton's method with idx equal to four.

series.**test_double_double_apollonius_at_series**(*vrblvl=0*)

Tests Newton's method starting at a series for the problem of Apollonius, in double double precision. The parameter t is the fourth variable, whence we call Newton's method with idx equal to four.

series.**test_double_double_pade**(*vrblvl=0*)

The function f(z) = ((1 + 1/2*z)/(1 + 2*z))^(1/2) is a solution x(s) of (1-s)*(x^2 - 1) + s*(3*x^2 - 3/2) = 0 Tests the Pade approximants in double double precision.

series.**test_double_double_viviani_at_point**(*vrblvl=0*)

Returns the system which stores the Viviani curve, with some solutions intersected with a plane, in double double precision.

series.**test_double_double_viviani_at_series**(*vrblvl=0*)

Computes the power series expansion for the Viviani curve, from a natural parameter perspective, in double double precision.

series.**test_double_pade**(*vrblvl=0*)

The function f(z) = ((1 + 1/2*z)/(1 + 2*z))^(1/2) is a solution x(s) of (1-s)*(x^2 - 1) + s*(3*x^2 - 3/2) = 0 Tests the Pade approximants in double precision.

series.**test_double_viviani_at_point**(*vrblvl=0*)

Returns the system which stores the Viviani curve, with some solutions intersected with a plane, in double precision.

series.**test_double_viviani_at_series**(*vrblvl=0*)

Computes the power series expansion for the Viviani curve, from a natural parameter perspective, in double precision.

series.**test_quad_double_apollonius_at_series**(*vrblvl=0*)

Tests Newton's method starting at a series for the problem of Apollonius, in quad double precision. The parameter t is the fourth variable, whence we call Newton's method with idx equal to four.

series.**test_quad_double_pade**(*vrblvl=0*)

The function f(z) = ((1 + 1/2*z)/(1 + 2*z))^(1/2) is a solution x(s) of (1-s)*(x^2 - 1) + s*(3*x^2 - 3/2) = 0 Tests the Pade approximants in quad double precision.

series.**test_quad_double_viviani_at_point**(*vrblvl=0*)

Returns the system which stores the Viviani curve, with some solutions intersected with a plane, in quad double precision.

series.**test_quad_double_viviani_at_series**(*vrblvl=0*)

Computes the power series expansion for the Viviani curve, from a natural parameter perspective, in quad double precision.

4.3.7 functions in the module curves

The module curves exports functions to approximate algebraic space curves with rational expressions, also known as Pade approximants, for use in a path tracker with apriori step size control.

curves.**clear_double_data**(*vrblvl=0*)

Clears the data used by the tracker in double precision.

curves.**clear_double_double_data**(*vrblvl=0*)

Clears the data used by the tracker in double double precision.

curves.**clear_quad_double_data**(*vrblvl=0*)

Clears the data used by the tracker in quad double precision.

curves.**double_closest_pole**(*vrblvl=0*)

Returns a tuple with the real and imaginary part of the closest pole used in the predictor in double precision.

curves.**double_double_closest_pole**(*vrblvl=0*)

Returns a tuple with the real and imaginary part of the closest pole used in the predictor in double double precision.

curves.**double_double_estimated_distance**(*vrblvl=0*)

Returns the estimated distance to the closest solution computed by the tracker in double double precision.

curves.**double_double_hessian_step**(*vrblvl=0*)

Returns the Hessian step in the tracker in double double precision.

curves.**double_double_pade_coefficients**(*idx*, *vrblvl=0*)

Returns a tuple of lists with the coefficients of the Pade approximants computed by the predictor in double double precision. The first list in the tuple holds the coefficients of the numerator, the second list in the tuple holds the denominator coefficients. On entry in idx is the index of a variable. The double coefficients on return are the highest parts of the double doubles.

curves.**double_double_pade_vector**(*dim*, *vrblvl=0*)

Returns the list of all coefficients over all dim variables, computed by the predictor in double double precision.

curves.**double_double_pole_radius**(*vrblvl=0*)

Returns the smallest pole radius, used in the predictor in double double precision.

curves.**double_double_pole_step**(*vrblvl=0*)

Returns the pole step in the tracker in double double precision.

curves.**double_double_poles**(*dim*, *vrblvl=0*)

Returns a list of lists of all poles of the vector of length dim, computed by the predictor in double double precision. The doubles on return are the highest parts of the double doubles.

curves.**double_double_predict_correct**(*vrblvl=0*)

Performs one predictor and one corrector step on the set homotopy and the set solution, in double double precision. If vrblvl > 0, then extra output is written.

curves.**double_double_series_coefficients**(*dim*, *vrblvl=0*)

Returns a list of lists with the coefficients of the series computed by the predictor in double double precision. The double coefficients are the highest parts of the double doubles. On entry in dim is the number of variables.

curves.**double_double_step_size**(*vrblvl=0*)

Returns the step size in the tracker in double double precision.

curves.**double_double_t_value**(*vrblvl=0*)

Returns the current t value in the tracker in double double precision.

curves.**double_double_track**(*target*, *start*, *startsols*, *filename=''*, *mhom=0*, *partition=None*, *vrblvl=0*)

Does path tracking in double double precision. On input are a target system, a start system with solutions. The *target* is a list of strings representing the polynomials of the target system (which has to be solved). The *start* is a list of strings representing the polynomials of the start system, with known solutions in sols. The *startsols* is a list of strings representing start solutions. The first optional argument is the *filename* for writing extra diagnostics during the tracking. By default *mhom* is zero and all happens in the original coordinates, if *mhom* equals one, then 1-homogeneous coordinates are used, and if *mhom* is two or higher, then multi-homogenization applies and *partition* contains the index representation of the partition of the set of variables. This index representation is a list of as many indices as the number of variables, defining which set of the partition each variables belongs to. On return is a tuple, with first the gamma used in the homotopy and then second, the string representations of the solutions computed at the end of the paths. Note: for mhom > 0 to work, the target, start system and solution must be provided in homogeneneous coordinates.

curves.**double_estimated_distance**(*vrblvl=0*)

Returns the estimated distance to the closest solution computed by the tracker in double precision.

curves.**double_hessian_step**(*vrblvl=0*)

Returns the Hessian step in the tracker in double precision.

curves.**double_pade_coefficients**(*idx*, *vrblvl=0*)

Returns a tuple of lists with the coefficients of the Pade approximants computed by the predictor in double precision. The first list in the tuple holds the coefficients of the numerator, the second list in the tuple holds the denominator coefficients. On entry in idx is the index of a variable.

curves.**double_pade_vector**(*dim*, *vrblvl=0*)

Returns the list of all coefficients over all dim variables, computed by the predictor in double precision.

curves.**double_pole_radius**(*vrblvl=0*)

Returns the smallest pole radius, used in the predictor in double precision.

curves.**double_pole_step**(*vrblvl=0*)

Returns the pole step in the tracker in double precision.

curves.**double_poles**(*dim*, *vrblvl=0*)

Returns a list of lists of all poles of the vector of length dim, computed by the predictor in double precision.

curves.**double_predict_correct**(*vrblvl=0*)

Performs one predictor and one corrector step on the set homotopy and the set solution, in double precision. If vrblvl > 0, then extra output is written.

curves.**double_series_coefficients**(*dim*, *vrblvl=0*)

Returns a list of lists with the coefficients of the series computed by the predictor in double precision. On entry in dim is the number of variables.

curves.**double_step_size**(*vrblvl=0*)

Returns the step size in the tracker in double precision.

curves.**double_t_value**(*vrblvl=0*)

Returns the current t value in the tracker in double precision.

curves.**double_track**(*target*, *start*, *startsols*, *filename=''*, *mhom=0*, *partition=None*, *vrblvl=0*)

Does path tracking in double precision. On input are a target system, a start system with solutions. The *target* is a list of strings representing the polynomials of the target system (which has to be solved). The *start* is a list of strings representing the polynomials of the start system, with known solutions in sols. The *startsols* is a list of strings representing start solutions. The first optional argument is the *filename* for writing extra diagnostics during the tracking. By default *mhom* is zero and all happens in the original coordinates, if *mhom* equals one, then 1-homogeneous coordinates are used, and if *mhom* is two or higher, then multi-homogenization applies and *partition* contains the index representation of the partition of the set of variables. This index representation is a list of as many indices as the number of variables, defining which set of the partition each variables belongs to. On return is a tuple, with first the gamma used in the homotopy and then second, the string representations of the solutions computed at the end of the paths. Note: for mhom > 0 to work, the target, start system and solution must be provided in homogeneneous coordinates.

curves.**get_corrector_residual_tolerance**(*vrblvl=0*)

Returns the current tolerance on the corrector residual. The corrector stops if the residual of the current approximation drops below this tolerance.

curves.**get_curvature_beta_factor**(*vrblvl=0*)

Returns the current multiplication factor of the curvature bound. This curvature bound gives an upper bound on a safe step size. The step size is set by multiplication of the curvature bound with the beta factor.

curves.**get_degree_of_denominator**(*vrblvl=0*)

Returns the current value of the degree of the denominator of the Pade approximant, evaluated to predict the next solution on a path.

curves.**get_degree_of_numerator**(*vrblvl=0*)

Returns the current value of the degree of the numerator of the Pade approximant, evaluated to predict the next solution on a path.

curves.**get_double_double_predicted_solution**(*vrblvl=0*)

Returns the predicted solution on the path, in double double precision, which starts at the solution set with set_double_double_solution(). If vrblvl > 0, then extra output is written.

curves.**get_double_double_solution**(*vrblvl=0*)

Returns the current solution on the path, in double double precision, which starts at the solution set with set_double_double_solution(). If vrblvl > 0, then extra output is written.

curves.**get_double_predicted_solution**(*vrblvl=0*)

Returns the predicted solution on the path, in double precision, which starts at the solution set with set_double_solution(). If vrblvl > 0, then extra output is written.

curves.**get_double_solution**(*vrblvl=0*)

Returns the current solution on the path, in double precision, which starts at the solution set with set_double_solution(). If vrblvl > 0, then extra output is written.

curves.**get_gamma_constant**(*vrblvl=0*)

Returns the current value of the gamma constant in the homotopy. A random value for gamma will guarantee the absence of singular solutions along a path, as unlucky choices belong to an algebraic set. A tuple of two floats is returned, respectively with the real and imaginary parts of the complex value for the gamma constant.

curves.**get_maximum_corrector_steps**(*vrblvl=0*)

Returns the current value of the maximum number of corrector steps executed after the predictor stage.

curves.**get_maximum_step_size**(*vrblvl=0*)

Returns the current value of the maximum step size. The step size is the increment to the continuation parameter.

curves.**get_maximum_steps_on_path**(*vrblvl=0*)

Returns the current value of the maximum number of steps on a path. The path trackers abandons the tracking of a path once the number of steps reaches this maximum number.

curves.**get_minimum_step_size**(*vrblvl=0*)

Returns the current value of the minimum step size. The path tracking will stop if the step size is larger than the minimum step size and if the predictor residual is larger than the value for alpha, the tolerance on the predictor residual.

curves.**get_parameter_value**(*idx*, *vrblvl=0*)

Returns the value of the parameter with index idx, where idx is an integer in 1, 2, .., 12. The verbose level is given by vrblvl.

curves.**get_pole_radius_beta_factor**(*vrblvl=0*)

Returns the current multiplication factor of the smallest pole radius. The smallest radius of the poles of the Pade approximant gives an upper bound on a safe step size. The step size is set by multiplication of the smallest pole radius with the beta factor.

curves.**get_predictor_residual_alpha**(*vrblvl=0*)

Returns the current tolerance on the residual of the predictor. This alpha parameter controls the accuracy of the tracking. As long as the residual of the evaluated predicted solution is larger than alpha, the step size is cut in half.

curves.**get_quad_double_predicted_solution**(*vrblvl=0*)

Returns the predicted solution on the path, in quad double precision, which starts at the solution set with set_quad_double_solution(). If vrblvl > 0, then extra output is written.

curves.**get_quad_double_solution**(*vrblvl=0*)

Returns the current solution on the path, in quad double precision, which starts at the solution set with set_quad_double_solution(). If vrblvl > 0, then extra output is written.

curves.**get_zero_series_coefficient_tolerance**(*vrblvl=0*)

Returns the current tolerance on the series coefficient to be zero. A coefficient in a power series will be considered as zero if its absolute value drops below this tolerance.

curves.**initialize_double_artificial_homotopy**(*target*, *start*, *homogeneous=False*, *vrblvl=0*)

Initializes the homotopy with the target and start system for a step-by-step run of the series-Pade tracker, in double precision. If homogeneous, then path tracking happens in projective space, otherwise the original affine coordinates are used. If vrblvl > 0, then extra output is written. Returns the failure code of the homotopy initializer.

curves.**initialize_double_double_artificial_homotopy**(*target*, *start*, *homogeneous=False*, *vrblvl=0*)

Initializes the homotopy with the target and start system for a step-by-step run of the series-Pade tracker, in double double precision. If homogeneous, then path tracking happens in projective space, otherwise the original affine coordinates are used. If vrblvl > 0, then extra output is written. Returns the failure code of the homotopy initializer.

curves.**initialize_double_double_parameter_homotopy**(*hom*, *idx*, *vrblvl=0*)

Initializes the homotopy with the polynomials in hom for a step-by-step run of the series-Pade tracker, in double double precision. The value idx gives the index of the continuation parameter and is the index of one of the variables in the homotopy hom. If vrblvl > 0, then extra output is written. Returns the failure code of the homotopy initializer.

curves.**initialize_double_parameter_homotopy**(*hom*, *idx*, *vrblvl=0*)

Initializes the homotopy with the polynomials in hom for a step-by-step run of the series-Pade tracker, in double precision. The value idx gives the index of the continuation parameter and is the index of one of the variables in the homotopy hom. If vrblvl > 0, then extra output is written. Returns the failure code of the homotopy initializer.

curves.**initialize_quad_double_artificial_homotopy**(*target*, *start*, *homogeneous=False*, *vrblvl=0*)

Initializes the homotopy with the target and start system for a step-by-step run of the series-Pade tracker, in quad double precision. If homogeneous, then path tracking happens in projective space, otherwise the original affine coordinates are used. If vrblvl > 0, then extra output is written. Returns the failure code of the homotopy initializer.

curves.**initialize_quad_double_parameter_homotopy**(*hom*, *idx*, *vrblvl=0*)

Initializes the homotopy with the polynomials in hom for a step-by-step run of the series-Pade tracker, in quad double precision. The value idx gives the index of the continuation parameter and is the index of one of the variables in the homotopy hom. If vrblvl > 0, then extra output is written. Returns the failure code of the homotopy initializer.

curves.**main**()

Runs some tests on tuning and tracking.

curves.**next_double_double_loop**(*hom*, *idx*, *sols*, *interactive=False*, *vrblvl=0*)

Runs the series-Pade tracker step by step in double double precision. On input is a natural parameter homotopy with solutions. The *hom* is a list of strings representing the polynomials of the natural parameter homotopy. The *idx* is the index of the variable in hom which is the continuation parameter. The *sols* is a list of strings representing start solutions. The start solutions do *not* contain the value of the continuation parameter, which is assumed to be equal to zero. Prompts before each predictor-corrector step, if *interactive*. If vrblvl > 0, then extra output is written. On return are the string representations of the solutions computed at the end of the paths.

curves.**next_double_double_track**(*target*, *start*, *sols*, *homogeneous=False*, *interactive=False*, *vrblvl=0*)

Runs the series-Pade tracker step by step in double double precision, for an artificial-parameter homotopy. On input are a target system and a start system with solutions. The *target* is a list of strings representing the polynomials of the target system (which has to be solved). The *start* is a list of strings representing the polynomials of the start system, with known solutions in *sols*. The *sols* is a list of strings representing start solutions. Prompts for each predictor-corrector step, if *interactive*. If *vrblvl* > 0, then extra output is written. If *homogeneous*, then

path tracking happens in projective space, otherwise the original affine coordinates are used. On return are the string representations of the solutions computed at the end of the paths.

curves.**next_double_loop**(*hom*, *idx*, *sols*, *interactive=False*, *vrblvl=0*)

Runs the series-Pade tracker step by step in double precision. On input is a natural parameter homotopy with solutions. The *hom* is a list of strings representing the polynomials of the natural parameter homotopy. The *idx* is the index of the variable in hom which is the continuation parameter. The *sols* is a list of strings representing start solutions. The start solutions do *not* contain the value of the continuation parameter, which is assumed to be equal to zero. Prompts before each predictor-corrector step, if *interactive*. If vrblvl > 0, then extra output is written. On return are the string representations of the solutions computed at the end of the paths.

curves.**next_double_track**(*target*, *start*, *sols*, *homogeneous=False*, *interactive=False*, *vrblvl=0*)

Runs the series-Pade tracker step by step in double precision, for an artificial-parameter homotopy. On input are a target system and a start system with solutions. The *target* is a list of strings representing the polynomials of the target system (which has to be solved). The *start* is a list of strings representing the polynomials of the start system, with known solutions in *sols*. The *sols* is a list of strings representing start solutions. Prompts for each predictor-corrector step, if *interactive*. If *vrblvl* > 0, then extra output is written. If *homogeneous*, then path tracking happens in projective space, otherwise the original affine coordinates are used. On return are the string representations of the solutions computed at the end of the paths.

curves.**next_quad_double_loop**(*hom*, *idx*, *sols*, *interactive=False*, *vrblvl=0*)

Runs the series-Pade tracker step by step in quad double precision. On input is a natural parameter homotopy with solutions. The *hom* is a list of strings representing the polynomials of the natural parameter homotopy. The *idx* is the index of the variable in hom which is the continuation parameter. The *sols* is a list of strings representing start solutions. The start solutions do *not* contain the value of the continuation parameter, which is assumed to be equal to zero. Prompts before each predictor-corrector step, if *interactive*. If vrblvl > 0, then extra output is written. On return are the string representations of the solutions computed at the end of the paths.

curves.**next_quad_double_track**(*target*, *start*, *sols*, *homogeneous=False*, *interactive=False*, *vrblvl=0*)

Runs the series-Pade tracker step by step in quad double precision, for an artificial-parameter homotopy. On input are a target system and a start system with solutions. The *target* is a list of strings representing the polynomials of the target system (which has to be solved). The *start* is a list of strings representing the polynomials of the start system, with known solutions in *sols*. The *sols* is a list of strings representing start solutions. Prompts for each predictor-corrector step, if *interactive*. If *vrblvl* > 0, then extra output is written. If *homogeneous*, then path tracking happens in projective space, otherwise the original affine coordinates are used. On return are the string representations of the solutions computed at the end of the paths.

curves.**quad_double_closest_pole**(*vrblvl=0*)

Returns a tuple with the real and imaginary part of the closest pole used in the predictor in quad double precision.

curves.**quad_double_estimated_distance**(*vrblvl=0*)

Returns the estimated distance to the closest solution computed by the tracker in quad double precision.

curves.**quad_double_hessian_step**(*vrblvl=0*)

Returns the Hessian step in the tracker in quad double precision.

curves.**quad_double_pade_coefficients**(*idx*, *vrblvl=0*)

Returns a tuple of lists with the coefficients of the Pade approximants computed by the predictor in quad double precision. The first list in the tuple holds the coefficients of the numerator, the second list in the tuple holds the denominator coefficients. On entry in idx is the index of a variable. The double coefficients on return are the highest parts of the quad doubles.

curves.**quad_double_pade_vector**(*dim*, *vrblvl=0*)

Returns the list of all coefficients over all dim variables, computed by the predictor in quad double precision.

curves.**quad_double_pole_radius**(*vrblvl=0*)

Returns the smallest pole radius, used in the predictor in quad double precision.

curves.**quad_double_pole_step**(*vrblvl=0*)

Returns the pole step in the tracker in quad double precision.

curves.**quad_double_poles**(*dim*, *vrblvl=0*)

Returns a list of lists of all poles of the vector of length dim, computed by the predictor in quad double precision. The doubles on return are the highest parts of the quad doubles.

curves.**quad_double_predict_correct**(*vrblvl=0*)

Performs one predictor and one corrector step on the set homotopy and the set solution, in quad double precision. If vrblvl > 0, then extra output is written.

curves.**quad_double_series_coefficients**(*dim*, *vrblvl=0*)

Returns a list of lists with the coefficients of the series computed by the predictor in quad double precision. The double coefficients are the highest parts of the quad doubles. On entry in dim is the number of variables.

curves.**quad_double_step_size**(*vrblvl=0*)

Returns the step size in the tracker in quad double precision.

curves.**quad_double_t_value**(*vrblvl=0*)

Returns the current t value in the tracker in quad double precision.

curves.**quad_double_track**(*target*, *start*, *startsols*, *filename=''*, *mhom=0*, *partition=None*, *vrblvl=0*)

Does path tracking in quad double precision. On input are a target system, a start system with solutions. The *target* is a list of strings representing the polynomials of the target system (which has to be solved). The *start* is a list of strings representing the polynomials of the start system, with known solutions in sols. The *startsols* is a list of strings representing start solutions. The first optional argument is the *filename* for writing extra diagnostics during the tracking. By default *mhom* is zero and all happens in the original coordinates, if *mhom* equals one, then 1-homogeneous coordinates are used, and if *mhom* is two or higher, then multi-homogenization applies and *partition* contains the index representation of the partition of the set of variables. This index representation is a list of as many indices as the number of variables, defining which set of the partition each variables belongs to. On return is a tuple, with first the gamma used in the homotopy and then second, the string representations of the solutions computed at the end of the paths. Note: for mhom > 0 to work, the target, start system and solution must be provided in homogeneneous coordinates.

curves.**reset_parameters**(*precision=0*, *vrblvl=0*)

Resets the homotopy continuation parameters for the step-by-step path trackers, using the value of the precision, 0 for double, 1 for double double, or 2 for quad double.

curves.**set_corrector_residual_tolerance**(*tol*, *vrblvl=0*)

Sets the tolerance on the corrector residual to tol.

curves.**set_curvature_beta_factor**(*beta*, *vrblvl=0*)

Sets the curvature beta factor to the value of beta.

curves.**set_default_parameters**(*vrblvl=0*)

Sets the default values of the parameters. The verbose level is given by vrblvl.

curves.**set_degree_of_denominator**(*deg*, *vrblvl=0*)

Set the value of the degree of the denominator of the Pade approximant to deg. The verbose level is given by vrblvl.

curves.**set_degree_of_numerator**(*deg*, *vrblvl=0*)

Set the value of the degree of the numerator of the Pade approximant to deg. The verbose level is given by vrblvl.

curves.**set_double_double_solution**(*nvr*, *sol*, *vrblvl=0*)

Sets the start solution in *sol* for the step-by-step run of the series-Pade tracker, in double double precision. The number of variables is in *nvr*. If vrblvl > 0, then extra output is written.

curves.**set_double_solution**(*nvr*, *sol*, *vrblvl=0*)

Sets the start solution in *sol* for the step-by-step run of the series-Pade tracker, in double precision. The number of variables is in *nvr*. If vrblvl > 0, then extra output is written.

curves.**set_gamma_constant**(*gamma*, *vrblvl=0*)

Sets the gamma constant to the value of the complex number given by gamma. The verbose level is given by vrblvl.

curves.**set_maximum_corrector_steps**(*maxsteps*, *vrblvl=0*)

Sets the maximum corrector steps to maxsteps.

curves.**set_maximum_step_size**(*maxstep*, *vrblvl=0*)

Sets the maximum value of the step size to maxstep.

curves.**set_maximum_steps_on_path**(*maxsteps*, *vrblvl=0*)

Sets the maximum steps on the path to maxsteps.

curves.**set_minimum_step_size**(*minstep*, *vrblvl=0*)

Sets the minimum value of the step size to minstep.

curves.**set_parameter_value**(*idx*, *value*, *vrblvl=0*)

Sets the parameter with index idx to the given value. The verbose level is given by vrblvl.

curves.**set_pole_radius_beta_factor**(*beta*, *vrblvl=0*)

Sets the pole radius beta factor to the value of beta.

curves.**set_predictor_residual_alpha**(*alpha*, *vrblvl=0*)

Sets the tolerance on the residual of the predictor to alpha.

curves.**set_quad_double_solution**(*nvr*, *sol*, *vrblvl=0*)

Sets the start solution in *sol* for the step-by-step run of the series-Pade tracker, in quad double precision. The number of variables is in *nvr*. If vrblvl > 0, then extra output is written.

curves.**set_zero_series_coefficient_tolerance**(*tol*, *vrblvl=0*)

Sets the tolerance on the zero series coefficient to tol.

curves.**symbolic_pade_approximant**(*cff*)

Given in cff are the coefficients of numerator and denominator of a Pade approximant, given as a tuple of two lists. On return is the string representation of the Pade approximant, using 't' as the variable.

curves.**symbolic_pade_vector**(*cff*)

Given in cff are the coefficients of numerator and denominator of a Pade vector, given as a list of tuples of two lists each. On return is the string representation of the Pade approximant, using 't' as the variable.

curves.**test_double_double_hyperbola**(*vrblvl=0*)

Tests the step-by-step Pade tracker on a hyperbola, in double double precision.

curves.**test_double_double_track**(*vrblvl=0*)

Runs on the mickey mouse example of two quadrics, in double double precision.

curves.**test_double_hyperbola**(*vrblvl=0*)

Tests the step-by-step Pade tracker on a hyperbola, in double precision.

curves.**test_double_track**(*vrblvl=0*)

Runs on the mickey mouse example of two quadrics, in double precision.

curves.**test_next_double_double_track**(*vrblvl=0*)

Runs on the mickey mouse example of two quadrics, with a step-by-step tracker in double double precision.

curves.**test_next_double_track**(*vrblvl=0*)

Runs on the mickey mouse example of two quadrics, with a step-by-step tracker in double precision.

curves.**test_next_quad_double_track**(*vrblvl=0*)

Runs on the mickey mouse example of two quadrics, with a step-by-step tracker in quad double precision.

curves.**test_quad_double_hyperbola**(*vrblvl=0*)

Tests the step-by-step Pade tracker on a hyperbola, in quad double precision.

curves.**test_quad_double_track**(*vrblvl=0*)

Runs on the mickey mouse example of two quadrics, in quad double precision.

curves.**test_tuning**(*vrblvl=0*)

Tests the tuning of the parameters.

curves.**write_parameters**(*vrblvl=0*)

Writes the values of the homotopy continuation parameters.

4.3.8 functions in the module deflation

By random choices of complex constants in an artificial parameter homotopy, singularities along a path are avoided. At the end of solution paths, we may encounter isolated singular solutions. The method of deflation is often effective at accurately computing isolated singular solutions.

Deflation restores the quadratic convergence of Newton's method at a singular, isolated solution.

deflation.**double_deflate**(*pols*, *sols*, *maxitr=3*, *maxdef=3*, *tolerr=1e-08*, *tolres=1e-08*, *tolrnk=1e-06*, *vrblvl=0*)

Applies deflation on the solutions sols of the system in pols, in double precision. The parameters are as follows:

maxitr: the maximum number of iterations per root,

maxdef: the maximum number of deflations per root,

tolerr: tolerance on the forward error on each root,

tolres: tolerance on the backward error on each root,

tolrnk: tolerance on the numerical rank of the Jacobian matrices.

deflation.**double_double_deflate**(*pols*, *sols*, *maxitr=3*, *maxdef=3*, *tolerr=1e-08*, *tolres=1e-08*, *tolrnk=1e-06*, *vrblvl=0*)

Applies deflation on the solutions sols of the system in pols, in double double precision. The parameters are as follows:

maxitr: the maximum number of iterations per root,

maxdef: the maximum number of deflations per root,

tolerr: tolerance on the forward error on each root,

tolres: tolerance on the backward error on each root,

tolrnk: tolerance on the numerical rank of the Jacobian matrices.

deflation.**double_double_multiplicity**(*system*, *solution*, *order=5*, *tol=1e-08*, *vrblvl=0*)

Computes the multiplicity structure in double double precision of an isolated solution (in the string *solution*) of a polynomial system (in the list *system*). The other parameters are

order: the maximum order of differentiation,

tol: tolerance on the numerical rank,

vrblvl: is the verbose level.

On return is the computed multiplicity.

deflation.**double_double_newton_step**(*pols*, *sols*, *vrblvl=0*)

Applies one Newton step to the *sols* of the *pols*, in double double precision.

deflation.**double_multiplicity**(*system*, *solution*, *order=5*, *tol=1e-08*, *vrblvl=0*)

Computes the multiplicity structure in double precision of an isolated solution (in the string *solution*) of a polynomial system (in the list *system*). The other parameters are

order: the maximum order of differentiation,

tol: tolerance on the numerical rank,

vrblvl: is the verbose level.

On return is the computed multiplicity.

deflation.**double_newton_step**(*pols*, *sols*, *vrblvl=0*)

Applies one Newton step to the *sols* of the *pols*, in double precision.

deflation.**main**()

Runs some tests.

deflation.**quad_double_deflate**(*pols*, *sols*, *maxitr=3*, *maxdef=3*, *tolerr=1e-08*, *tolres=1e-08*, *tolrnk=1e-06*, *vrblvl=0*)

Applies deflation on the solutions sols of the system in pols, in quad double precision. The parameters are as follows:

maxitr: the maximum number of iterations per root,

maxdef: the maximum number of deflations per root,

tolerr: tolerance on the forward error on each root,

tolres: tolerance on the backward error on each root,

tolrnk: tolerance on the numerical rank of the Jacobian matrices.

deflation.**quad_double_multiplicity**(*system*, *solution*, *order=5*, *tol=1e-08*, *vrblvl=0*)

Computes the multiplicity structure in quad double precision of an isolated solution (in the string *solution*) of a polynomial system (in the list *system*). The other parameters are

order: the maximum order of differentiation,

tol: tolerance on the numerical rank,

vrblvl: is the verbose level.

On return is the computed multiplicity.

deflation.**quad_double_newton_step**(*pols*, *sols*, *vrblvl=0*)

Applies one Newton step to the *sols* of the *pols*, in quad double precision.

deflation.**test_double_deflate**(*vrblvl=0*)

Tests deflation in double precision, on the 'ojika1' demonstration test system, in the paper by T. Ojika: "Modified deflation algorithm for the solution of singular problems. I. A system of nonlinear algebraic equations." J. Math. Anal. Appl. 123, 199-221, 1987.

deflation.**test_double_double_deflate**(*vrblvl=0*)

Tests deflation in double double precision, on the 'ojika1' demonstration test system, in the paper by T. Ojika: "Modified deflation algorithm for the solution of singular problems. I. A system of nonlinear algebraic equations." J. Math. Anal. Appl. 123, 199-221, 1987.

deflation.**test_double_double_multiplicity**(*vrblvl=0*)

Tests the multiplicity structure in double double precision.

deflation.**test_double_double_newton_step**(*vrblvl=0*)

Tests Newton's method in double double precision.

deflation.**test_double_multiplicity**(*vrblvl=0*)

Tests the multiplicity structure in double precision.

deflation.**test_double_newton_step**(*vrblvl=0*)

Tests Newton's method in double precision.

deflation.**test_quad_double_deflate**(*vrblvl=0*)

Tests deflation in quad double precision, on the 'ojika1' demonstration test system, in the paper by T. Ojika: "Modified deflation algorithm for the solution of singular problems. I. A system of nonlinear algebraic equations." J. Math. Anal. Appl. 123, 199-221, 1987.

deflation.**test_quad_double_multiplicity**(*vrblvl=0*)

Tests the multiplicity structure in quad double precision.

deflation.**test_quad_double_newton_step**(*vrblvl=0*)

Tests Newton's method in quad double precision.

4.4 homotopies for enumerative geometry

Homotopy methods to two types of problems in enumerative geometry are provided, applying Pieri and Littlewood-Richardson root counts.

4.4.1 functions in the module schubert

Numerical Schubert calculus defines homotopies for enumerative geometry.

schubert.**double_littlewood_richardson_homotopies**(*ndim*, *kdim*, *brackets*, *verbose=True*, *vrfcnd=False*, *minrep=True*, *tosqr=False*, *outputfilename=''*, *vrblvl=0*)

In n-dimensional space we consider k-dimensional planes, subject to intersection conditions represented by brackets. The parameters *ndim* and *kdim* give values for n and k respectively. The parameter brackets is a list of brackets. A bracket is a list of as many natural numbers (in the range 1..*ndim*) as *kdim*. The Littlewood-Richardson homotopies compute k-planes that meet the flags at spaces of dimensions prescribed by the brackets, in standard double precision. Four options are passed as Booleans:

verbose: for adding extra output during computations,

vrfcnd: for extra diagnostic verification of Schubert conditions,

minrep: for a minimial representation of the problem formulation,

tosqr: to square the overdetermined systems.

On return is a 4-tuple. The first item of the tuple is the formal root count, sharp for general flags, then as second item the coordinates of the flags. The coordinates of the flags are stored row wise in a list of real and imaginary

parts. The third and fourth item of the tuple on return are respectively the polynomial system that has been solved and its solutions. The length of the list of solution should match the root count.

schubert.**main**()

Runs some tests.

schubert.**make_pieri_system**(*mdim*, *pdim*, *qdeg*, *planes*, *is_real=False*, *vrblvl=0*)

Makes the polynomial system defined by the mdim-planes in the list planes.

schubert.**pieri_localization_poset**(*mdim*, *pdim*, *qdeg=0*, *size=10240*, *vrblvl=0*)

Returns the string representation of the localization poset used to compute the Pieri root count, for the number of *pdim*-plane producing maps of degree *qdeg* that meet *mdim*-planes at mdim*pdim + qdeg*(mdim+pdim) points. The size of the string representation is given on input in size.

schubert.**pieri_root_count**(*mdim*, *pdim*, *qdeg=0*, *vrblvl=0*)

Computes the number of *pdim*-plane producing maps of degree *qdeg* that meet *mdim*-planes at mdim*pdim + qdeg*(mdim+pdim) points.

schubert.**random_complex_matrices**(*nbr*, *nbrows*, *nbcols*)

Returns a list of matrix of length *nbr*, all of dimension *nbrows* by *nbcols*.

schubert.**random_complex_matrix**(*nbrows*, *nbcols*)

Returns a random *nbrows*-by-*nbcols* matrix with randomly generated complex coefficients on the unit circle, as a list of *rows*.

schubert.**real_osculating_planes**(*mdim*, *pdim*, *qdeg=0*, *vrblvl=0*)

Returns m*p + qdeg*(m+p) real m-planes osculating a rational normal curve. The value of the verbose level is given by vrblvl.

schubert.**resolve_schubert_conditions**(*ndim*, *kdim*, *brackets*, *vrblvl=0*)

In n-dimensional space we consider k-dimensional planes, subject to intersection conditions represented by brackets. The *brackets* is a list of brackets. A bracket is a list of as many natural numbers (in the range 1..*ndim*) as *kdim*. On return is the formal root count, which is sharp for general flags. and the coordinates of the flags, stored row wise in a list of real and imaginary parts.

schubert.**run_pieri_homotopies**(*mdim*, *pdim*, *qdeg*, *planes*, *vrblvl=0*, **pts*)

Computes the number of *pdim*-plane producing maps of degree *qdeg* that meet *mdim*-planes at mdim*pdim + qdeq*(mdim+pdim) points. For *qdeg* = 0, there are no interpolation points in *pts*. The planes and pts are assumed to be complex.

schubert.**test_littlewood_richardson_homotopies**(*vrblvl=0*)

Performs a test on the Littlewood-Richardson homotopies.

schubert.**test_littlewood_richardson_rule**(*vrblvl=0*)

Tests the Littlewood-Richardson rule.

schubert.**test_pieri_count**(*vrblvl=0*)

Tests the Pieri root count.

schubert.**test_pieri_curves**(*vrblvl=0*)

Computes all line producing curves that meet given lines in 3-space, at given interpolation points, computed with Pieri homotopies.

schubert.**test_pieri_homotopies**(*vrblvl=0*)

Tests running the Pieri homotopies on the four lines problem.

schubert.**test_pieri_problem**(*vrblvl=0*)

Tests the making of a Pieri problem.

4.5 numerical irreducible decomposition

The second blackbox solver computes a numerical irreducible decomposition of the solution set of a polynomial system.

4.5.1 functions in the module sets

This module exports routines of PHCpack to manipulate positive dimensional solution sets of polynomial systems.

A numerical data structure to represent a pure dimensional solution set is a witness set, which consists of two parts:

1. the original polynomial system, augmented with as many random linear equations as the dimension of the set; and
2. isolated solutions of the augmented system, as many as the degree of the solution set.

The solutions of the augmented system are called witness points and are generic points on the algebraic set.

The embed functions add slack variables and random hyperplanes. The number of slack variables equals the number of random hyperplanes, which in turn equals the dimension of the solution set. The drop functions remove the added slack variables from the polynomials and the coordinates of the solutions. Given a witness set and a point, a homotopy membership determines whether the point belongs to the solution set represented by the witness set.

`sets.`**double_double_embed**(*nvr*, *topdim*, *pols*, *vrblvl=0*)

Given in *pols* a list of strings representing polynomials in *nvr* variables, with coefficients in double double precision, this function returns an embedding of *pols* of dimension *topdim*. The *topdim* is the top dimension which equals the expected highest dimension of a component of the solution set of the system of polynomials. If *vrblvl* is larger than 0, then extra information is printed.

`sets.`**double_double_hypersurface_set**(*nvr*, *hpol*, *vrblvl=0*)

Given in *hpol* the string representation of a polynomial in *nvar* variables (ending with a semicolon), on return is an embedded system and its solutions which represents a witness set for *hpol*. The number of solutions on return should equal the degree of the polynomial in *hpol*.

`sets.`**double_double_laurent_embed**(*nvr*, *topdim*, *pols*, *vrblvl=0*)

Given in *pols* a list of strings representing Laurent polynomials in *nvr* variables, with coefficients in double double precision, this function returns an embedding of *pols* of dimension *topdim*. The *topdim* is the top dimension which equals the expected highest dimension of a component of the solution set of the system of polynomials. If *vrblvl* is larger than 0, then the names of the procedures called in the running of the blackbox solver will be listed.

`sets.`**double_double_laurent_membertest**(*wsys*, *gpts*, *dim*, *point*, *evatol=1e-06*, *memtol=1e-06*, *tasks=0*, *vrblvl=0*)

Applies the homotopy membership test for a point to belong to a witness set of dimension *dim*, given by an embedded Laurent system in *wsys*, with corresponding generic points in *gpts*. The coordinates of the test point are given in the list *point*, as a list of doubles, with the real and imaginary part of each coordinate of the point. By default, *verbose* is True. The number of threads is given in *tasks*. If *tasks* is zero, then no multithreading is applied in the homotopy membership test. Calculations happen in double double precision. The default values for the evaluation (*evatol*) and the membership (*memtol*) allow for singular values at the end points of the paths in the homotopy membership test.

`sets.`**double_double_membertest**(*wsys*, *gpts*, *dim*, *point*, *evatol=1e-06*, *memtol=1e-06*, *tasks=0*, *vrblvl=0*)

Applies the homotopy membership test for a point to belong to a witness set of dimension *dim*, given by an embedding polynomial system in *wsys*, with corresponding generic points in *gpts*. The coordinates of the test point are given in the list *point*, as a list of doubles, with the real and imaginary part of each coordinate of the point. By default, *verbose* is True. The number of threads is given in *tasks*. If *tasks* is zero, then no multithreading is applied in the homotopy membership test. Calculations happen in double double precision. The default values

for the evaluation (*evatol*) and the membership (*memtol*) allow for singular values at the end points of the paths in the homotopy membership test.

sets.**double_embed**(*nvr*, *topdim*, *pols*, *vrblvl=0*)

Given in *pols* a list of strings representing polynomials in *nvr* variables, with coefficients in double precision, this function returns an embedding of *pols* of dimension *topdim*. The *topdim* is the top dimension which equals the expected highest dimension of a component of the solution set of the system of polynomials. If *vrblvl* is larger than 0, then extra information is printed.

sets.**double_hypersurface_set**(*nvr*, *hpol*, *vrblvl=0*)

Given in *hpol* the string representation of a polynomial in *nvar* variables (ending with a semicolon), on return is an embedded system and its solutions which represents a witness set for *hpol*. The number of solutions on return should equal the degree of the polynomial in *hpol*.

sets.**double_laurent_embed**(*nvr*, *topdim*, *pols*, *vrblvl=0*)

Given in *pols* a list of strings representing Laurent polynomials in *nvr* variables, with coefficients in double precision, this function returns an embedding of *pols* of dimension *topdim*. The *topdim* is the top dimension which equals the expected highest dimension of a component of the solution set of the system of polynomials. If *vrblvl* is larger than 0, then the names of the procedures called in the running of the blackbox solver will be listed.

sets.**double_laurent_membertest**(*wsys*, *gpts*, *dim*, *point*, *evatol=1e-06*, *memtol=1e-06*, *tasks=0*, *vrblvl=0*)

Applies the homotopy membership test for a point to belong to a witness set of dimension *dim*, given by an embedded Laurent system in *wsys*, with corresponding generic points in *gpts*. The coordinates of the test point are given in the list *point*, as a list of doubles, with the real and imaginary part of each coordinate of the point. By default, *verbose* is True. The number of threads is given in *tasks*. If *tasks* is zero, then no multithreading is applied in the homotopy membership test. Calculations happen in double precision. The default values for the evaluation (*evatol*) and the membership (*memtol*) allow for singular values at the end points of the paths in the homotopy membership test.

sets.**double_membertest**(*wsys*, *gpts*, *dim*, *point*, *evatol=1e-06*, *memtol=1e-06*, *tasks=0*, *vrblvl=0*)

Applies the homotopy membership test for a point to belong to a witness set of dimension *dim*, given by an embedding polynomial system in *wsys*, with corresponding generic points in *gpts*. The coordinates of the test point are given in the list *point*, as a list of doubles, with the real and imaginary part of each coordinate of the point. By default, *verbose* is True. The number of threads is given in *tasks*. If *tasks* is zero, then no multithreading is applied in the homotopy membership test. Calculations happen in double precision. The default values for the evaluation (*evatol*) and the membership (*memtol*) allow for singular values at the end points of the paths in the homotopy membership test.

sets.**drop_coordinate_from_double_double_solutions**(*sols*, *nvr*, *svar*, *vrblvl=0*)

Removes the variable with symbol in the string *svar* from the list *sols* of strings that represent solutions in *nvr* variables, in double double precision.

sets.**drop_coordinate_from_double_solutions**(*sols*, *nvr*, *svar*, *vrblvl=0*)

Removes the variable with symbol in the string *svar* from the list *sols* of strings that represent solutions in *nvr* variables, in double precision.

sets.**drop_coordinate_from_quad_double_solutions**(*sols*, *nvr*, *svar*, *vrblvl=0*)

Removes the variable with symbol in the string *svar* from the list *sols* of strings that represent solutions in *nvr* variables, in quad double precision.

sets.**drop_variable_from_double_double_laurent_polynomials**(*pols*, *svar*, *vrblvl=0*)

Removes the variable with symbol in the string *svar* from the list *pols* of strings that represent Laurent polynomials in several variables, with coefficients in double double precision. The system in *pols* must be square.

sets.**drop_variable_from_double_double_polynomials**(*pols*, *svar*, *vrblvl=0*)

Removes the variable with symbol in the string *svar* from the list *pols* of strings that represent polynomials in several variables, with coefficients in double double precision. The system in *pols* must be square.

sets.**drop_variable_from_double_laurent_polynomials**(*pols*, *svar*, *vrblvl=0*)

Removes the variable with symbol in the string *svar* from the list *pols* of strings that represent Laurent polynomials in several variables, with coefficients in double precision. The system in *pols* must be square.

sets.**drop_variable_from_double_polynomials**(*pols*, *svar*, *vrblvl=0*)

Removes the variable with symbol in the string *svar* from the list *pols* of strings that represent polynomials in several variables, with coefficients in double precision. The system in *pols* must be square.

sets.**drop_variable_from_quad_double_laurent_polynomials**(*pols*, *svar*, *vrblvl=0*)

Removes the variable with symbol in the string *svar* from the list *pols* of strings that represent Laurent polynomials in several variables, with coefficients in quad double precision. The system in *pols* must be square.

sets.**drop_variable_from_quad_double_polynomials**(*pols*, *svar*, *vrblvl=0*)

Removes the variable with symbol in the string *svar* from the list *pols* of strings that represent polynomials in several variables, with coefficients in quad double precision. The system in *pols* must be square.

sets.**drop_variable_from_table**(*name*, *vrblvl=0*)

Drops a variable with the given name from the symbol table. The verbose level is given by the value of vrblvl.

sets.**main**()

Runs some tests.

sets.**quad_double_embed**(*nvr*, *topdim*, *pols*, *vrblvl=0*)

Given in *pols* a list of strings representing polynomials in *nvr* variables, with coefficients in quad double precision, this function returns an embedding of *pols* of dimension *topdim*. The *topdim* is the top dimension which equals the expected highest dimension of a component of the solution set of the system of polynomials. If *vrblvl* is larger than 0, then extra information is printed.

sets.**quad_double_hypersurface_set**(*nvr*, *hpol*, *vrblvl=0*)

Given in *hpol* the string representation of a polynomial in *nvar* variables (ending with a semicolon), on return is an embedded system and its solutions which represents a witness set for *hpol*. The number of solutions on return should equal the degree of the polynomial in *hpol*.

sets.**quad_double_laurent_embed**(*nvr*, *topdim*, *pols*, *vrblvl=0*)

Given in *pols* a list of strings representing Laurent polynomials in *nvr* variables, with coefficients in quad double precision, this function returns an embedding of *pols* of dimension *topdim*. The *topdim* is the top dimension which equals the expected highest dimension of a component of the solution set of the system of polynomials. If *vrblvl* is larger than 0, then the names of the procedures called in the running of the blackbox solver will be listed.

sets.**quad_double_laurent_membertest**(*wsys*, *gpts*, *dim*, *point*, *evatol=1e-06*, *memtol=1e-06*, *tasks=0*, *vrblvl=0*)

Applies the homotopy membership test for a point to belong to a witness set of dimension *dim*, given by an embedded Laurent system in *wsys*, with corresponding generic points in *gpts*. The coordinates of the test point are given in the list *point*, as a list of doubles, with the real and imaginary part of each coordinate of the point. By default, *verbose* is True. The number of threads is given in *tasks*. If *tasks* is zero, then no multithreading is applied in the homotopy membership test. Calculations happen in quad double precision. The default values for the evaluation (*evatol*) and the membership (*memtol*) allow for singular values at the end points of the paths in the homotopy membership test.

sets.**quad_double_membertest**(*wsys*, *gpts*, *dim*, *point*, *evatol=1e-06*, *memtol=1e-06*, *tasks=0*, *vrblvl=0*)

Applies the homotopy membership test for a point to belong to a witness set of dimension *dim*, given by an embedding polynomial system in *wsys*, with corresponding generic points in *gpts*. The coordinates of the test

point are given in the list *point*, as a list of doubles, with the real and imaginary part of each coordinate of the point. By default, *verbose* is True. The number of threads is given in *tasks*. If *tasks* is zero, then no multithreading is applied in the homotopy membership test. Calculations happen in quad double precision. The default values for the evaluation (*evatol*) and the membership (*memtol*) allow for singular values at the end points of the paths in the homotopy membership test.

sets.**set_double_double_laurent_witness_set**(*nvr*, *dim*, *pols*, *sols*, *vrblvl=0*)

Given in *nvr* is the total number of variables in the list of Laurent polynomials in *pols* and its list of solutions in *sols*. The coefficients in the Laurent polynomials and the coordinates of the solutions will be parsed and stored in double double precision. The parameter *dim* equals the number of slack variables used in the embedding of *pols* and *sols*. This *dim* also equals the dimension of the solution set represented by the witness set given by the lists *pols* and *sols*. The symbols for the slack variables are swapped to the end of the symbol table in both the polynomials and the solutions.

sets.**set_double_double_witness_set**(*nvr*, *dim*, *pols*, *sols*, *vrblvl=0*)

Given in *nvr* is the total number of variables in the list of polynomials in *pols* and its list of solutions in *sols*. The coefficients in the polynomials and the coordinates of the solutions will be parsed and stored in double double precision. The parameter *dim* equals the number of slack variables used in the embedding of *pols* and *sols*. This *dim* also equals the dimension of the solution set represented by the witness set given by the lists *pols* and *sols*. The symbols for the slack variables are swapped to the end of the symbol table in both the polynomials and the solutions.

sets.**set_double_laurent_witness_set**(*nvr*, *dim*, *pols*, *sols*, *vrblvl=0*)

Given in *nvr* is the total number of variables in the list of Laurent polynomials in *pols* and its list of solutions in *sols*. The coefficients in the Laurent polynomials and the coordinates of the solutions will be parsed and stored in double precision. The parameter *dim* equals the number of slack variables used in the embedding of *pols* and *sols*. This *dim* also equals the dimension of the solution set represented by the witness set given by the lists *pols* and *sols*. The symbols for the slack variables are swapped to the end of the symbol table in both the polynomials and the solutions.

sets.**set_double_witness_set**(*nvr*, *dim*, *pols*, *sols*, *vrblvl=0*)

Given in *nvr* is the total number of variables in the list of polynomials in *pols* and its list of solutions in *sols*. The coefficients in the polynomials and the coordinates of the solutions will be parsed and stored in double precision. The parameter *dim* equals the number of slack variables used in the embedding of *pols* and *sols*. This *dim* also equals the dimension of the solution set represented by the witness set given by the lists *pols* and *sols*. The symbols for the slack variables are swapped to the end of the symbol table in both the polynomials and the solutions.

sets.**set_quad_double_laurent_witness_set**(*nvr*, *dim*, *pols*, *sols*, *vrblvl=0*)

Given in *nvr* is the total number of variables in the list of Laurent polynomials in *pols* and its list of solutions in *sols*. The coefficients in the Laurent polynomials and the coordinates of the solutions will be parsed and stored in quad double precision. The parameter *dim* equals the number of slack variables used in the embedding of *pols* and *sols*. This *dim* also equals the dimension of the solution set represented by the witness set given by the lists *pols* and *sols*. The symbols for the slack variables are swapped to the end of the symbol table in both the polynomials and the solutions.

sets.**set_quad_double_witness_set**(*nvr*, *dim*, *pols*, *sols*, *vrblvl=0*)

Given in *nvr* is the total number of variables in the list of polynomials in *pols* and its list of solutions in *sols*. The coefficients in the polynomials and the coordinates of the solutions will be parsed and stored in quad double precision. The parameter *dim* equals the number of slack variables used in the embedding of *pols* and *sols*. This *dim* also equals the dimension of the solution set represented by the witness set given by the lists *pols* and *sols*. The symbols for the slack variables are swapped to the end of the symbol table in both the polynomials and the solutions.

sets.**test_double_double_drop**(*vrblvl=0*)

Tests the removal of a slack variable in double double precision.

sets.**test_double_double_hypersurface_set**(*vrblvl=0*)

Tests the construction of a witness set of a hypersurface in double double precision.

sets.**test_double_double_laurent_twisted**(*vrblvl=0*)

Tests the computation of a witness set for the twisted cubic defined as a laurent system in double double precision. As the degree of the twisted cubic is three, the number of witness points must equal three as well.

sets.**test_double_double_member**(*vrblvl=0*)

Tests the membership in double double precision.

sets.**test_double_double_twisted**(*vrblvl=0*)

Tests the computation of a witness set for the twisted cubic in double double precision. As the degree of the twisted cubic is three, the number of witness points must equal three as well.

sets.**test_double_drop**(*vrblvl=0*)

Tests the removal of a slack variable in double precision.

sets.**test_double_hypersurface_set**(*vrblvl=0*)

Tests the construction of a witness set of a hypersurface in double precision.

sets.**test_double_laurent_twisted**(*vrblvl=0*)

Tests the computation of a witness set for the twisted cubic defined as a laurent system in double precision. As the degree of the twisted cubic is three, the number of witness points must equal three as well.

sets.**test_double_member**(*vrblvl=0*)

Tests the membership in double precision.

sets.**test_double_twisted**(*vrblvl=0*)

Tests the computation of a witness set for the twisted cubic in double precision. As the degree of the twisted cubic is three, the number of witness points must equal three as well.

sets.**test_quad_double_drop**(*vrblvl=0*)

Tests the removal of a slack variable in quad double precision.

sets.**test_quad_double_hypersurface_set**(*vrblvl=0*)

Tests the construction of a witness set of a hypersurface in quad double precision.

sets.**test_quad_double_laurent_twisted**(*vrblvl=0*)

Tests the computation of a witness set for the twisted cubic defined as a laurent system in quad double precision. As the degree of the twisted cubic is three, the number of witness points must equal three as well.

sets.**test_quad_double_member**(*vrblvl=0*)

Tests the membership in quad double precision.

sets.**test_quad_double_twisted**(*vrblvl=0*)

Tests the computation of a witness set for the twisted cubic in quad double precision. As the degree of the twisted cubic is three, the number of witness points must equal three as well.

4.5.2 functions in the module cascades

A cascade homotopy removes one hyperplane from an embedded system, taking the solutions with nonzero slack variables to solutions on lower dimensional components of the solution set of the original system.

cascades.**double_cascade_filter**(*dim*, *embpols*, *nonsols*, *tol*, *tasks=0*, *vrblvl=0*)

Runs one step in the cascade homotopy defined by the embedding of polynomials in *embpols*, starting at the solutions in *nonsols*, removing the last hyperplane from *embpols* at dimension *dim*. The tolerance *tol* is used to split filter the solutions. Computations happen in double precision. If *vrblvl* > 0, then extra output is written.

cascades.**double_cascade_step**(*dim*, *embsys*, *esols*, *tasks=0*, *vrblvl=0*)

Given in *embsys* an embedded polynomial system and solutions with nonzero slack variables in *esols*, does one step in the homotopy cascade, with double precision arithmetic. The dimension of the solution set represented by *embsys* and *esols* is the value of *dim*. The number of tasks in multithreaded path tracking is given by *tasks*. The default zero value of *tasks* indicates no multithreading. The list on return contains witness points on lower dimensional solution components.

cascades.**double_double_cascade_filter**(*dim*, *embpols*, *nonsols*, *tol*, *tasks=0*, *vrblvl=0*)

Runs one step in the cascade homotopy defined by the embedding of polynomials in *embpols*, starting at the solutions in *nonsols*, removing the last hyperplane from *embpols* at dimension *dim*. Computations happen in double double precision. The tolerance *tol* is used to split filter the solutions. If *vrblvl* > 0, then extra output is written.

cascades.**double_double_cascade_step**(*dim*, *embsys*, *esols*, *tasks=0*, *vrblvl=0*)

Given in *embsys* an embedded polynomial system and solutions with nonzero slack variables in *esols*, does one step in the homotopy cascade, with double double precision arithmetic. The dimension of the solution set represented by *embsys* and *esols* is the value of *dim*. The number of tasks in multithreaded path tracking is given by *tasks*. The default zero value of *tasks* indicates no multithreading. The list on return contains witness points on lower dimensional solution components.

cascades.**double_double_laurent_cascade_filter**(*dim*, *embpols*, *nonsols*, *tol*, *tasks=0*, *vrblvl=0*)

Runs one step in the cascade homotopy defined by the embedding of Laurent system in *embpols*, starting at the solutions in *nonsols*, removing the last hyperplane from *embpols* at dimension *dim*. The tolerance *tol* is used to split filter the solutions. Computations happen in double double precision. If *vrblvl* > 0, then extra output is written.

cascades.**double_double_laurent_cascade_step**(*dim*, *embsys*, *esols*, *tasks=0*, *vrblvl=0*)

Given in *embsys* an embedded Laurent polynomial system and solutions with nonzero slack variables in *esols*, does one step in the homotopy cascade, with double double precision arithmetic. The dimension of the solution set represented by *embsys* and *esols* is the value of *dim*. The number of tasks in multithreaded path tracking is given by *tasks*. The default zero value of *tasks* indicates no multithreading. The list on return contains witness points on lower dimensional solution components.

cascades.**double_double_laurent_top_cascade**(*nvr*, *dim*, *pols*, *tol=1e-06*, *tasks=0*, *vrblvl=0*)

Constructs an embedding of the Laurent polynomials in *pols*, with the number of variables in *pols* equal to *nvr*, where *dim* is the top dimension of the solution set. Applies the blackbox solver to the embedded system to compute in double double precision the generic points and the nonsolutions for use in the cascade. Returns a tuple with three items:

1. the embedded system,
2. the solutions with zero last coordinate w.r.t. *tol*,
3. the solutions with nonzero last coordinate w.r.t. *tol*.

The three parameters on input are

1. *tol* is the tolerance to decide whether a number is zero or not, used to split the solution list of the embedded system;

2. *tasks* is the number of tasks, 0 if no multitasking,
3. if *vrblvl* > 0, then extra output is written.

cascades.**double_double_top_cascade**(*nvr*, *dim*, *pols*, *tol=1e-06*, *tasks=0*, *vrblvl=0*)

Constructs an embedding of the polynomials in *pols*, with the number of variables in *pols* equal to *nvr*, where *dim* is the top dimension of the solution set. Applies the blackbox solver to the embedded system to compute in double double precision the generic points and the nonsolutions for use in the cascade. Returns a tuple with three items:

1. the embedded system,
2. the solutions with zero last coordinate w.r.t. *tol*,
3. the solutions with nonzero last coordinate w.r.t. *tol*.

The three parameters on input are

1. *tol* is the tolerance to decide whether a number is zero or not, used to split the solution list of the embedded system;
2. *tasks* is the number of tasks, 0 if no multitasking,
3. if *vrblvl* > 0, then extra output is written.

cascades.**double_laurent_cascade_filter**(*dim*, *embpols*, *nonsols*, *tol*, *tasks=0*, *vrblvl=0*)

Runs one step in the cascade homotopy defined by the embedding of Laurent system in *embpols*, starting at the solutions in *nonsols*, removing the last hyperplane from *embpols* at dimension *dim*. The tolerance *tol* is used to split filter the solutions. Computations happen in double precision. If *vrblvl* > 0, then extra output is written.

cascades.**double_laurent_cascade_step**(*dim*, *embsys*, *esols*, *tasks=0*, *vrblvl=0*)

Given in *embsys* an embedded Laurent polynomial system and solutions with nonzero slack variables in *esols*, does one step in the homotopy cascade, with double precision arithmetic. The dimension of the solution set represented by *embsys* and *esols* is the value of *dim*. The number of tasks in multithreaded path tracking is given by *tasks*. The default zero value of *tasks* indicates no multithreading. The list on return contains witness points on lower dimensional solution components.

cascades.**double_laurent_top_cascade**(*nvr*, *dim*, *pols*, *tol=1e-06*, *tasks=0*, *vrblvl=0*)

Constructs an embedding of the Laurent polynomials in *pols*, with the number of variables in *pols* equal to *nvr*, where *dim* is the top dimension of the solution set. Applies the blackbox solver to the embedded system to compute in double precision the generic points and the nonsolutions for use in the cascade. Returns a tuple with three items:

1. the embedded system,
2. the solutions with zero last coordinate w.r.t. *tol*,
3. the solutions with nonzero last coordinate w.r.t. *tol*.

The three parameters on input are

1. *tol* is the tolerance to decide whether a number is zero or not, used to split the solution list of the embedded system;
2. *tasks* is the number of tasks, 0 if no multitasking,
3. if *vrblvl* > 0, then extra output is written.

cascades.**double_top_cascade**(*nvr*, *dim*, *pols*, *tol=1e-06*, *tasks=0*, *vrblvl=0*)

Constructs an embedding of the polynomials in *pols*, with the number of variables in *pols* equal to *nvr*, where *dim* is the top dimension of the solution set. Applies the blackbox solver to the embedded system to compute in double precision the generic points and the nonsolutions for use in the cascade. Returns a tuple with three items:

1. the embedded system,
2. the solutions with zero last coordinate w.r.t. *tol*,
3. the solutions with nonzero last coordinate w.r.t. *tol*.

The three parameters on input are

1. *tol* is the tolerance to decide whether a number is zero or not, used to split the solution list of the embedded system;
2. *tasks* is the number of tasks, 0 if no multitasking,
3. if *vrblvl* > 0, then extra output is written.

cascades.**main**()

Runs some tests.

cascades.**quad_double_cascade_filter**(*dim*, *embpols*, *nonsols*, *tol*, *tasks=0*, *vrblvl=0*)

Runs one step in the cascade homotopy defined by the embedding of polynomials in *embpols*, starting at the solutions in *nonsols*, removing the last hyperplane from *embpols* at dimension *dim*. Computations happen in quad double precision. The tolerance *tol* is used to split filter the solutions. If *vrblvl* > 0, then extra output is written.

cascades.**quad_double_cascade_step**(*dim*, *embsys*, *esols*, *tasks=0*, *vrblvl=0*)

Given in *embsys* an embedded polynomial system and solutions with nonzero slack variables in *esols*, does one step in the homotopy cascade, with quad double precision arithmetic. The dimension of the solution set represented by *embsys* and *esols* is the value of *dim*. The number of tasks in multithreaded path tracking is given by *tasks*. The default zero value of *tasks* indicates no multithreading. The list on return contains witness points on lower dimensional solution components.

cascades.**quad_double_laurent_cascade_filter**(*dim*, *embpols*, *nonsols*, *tol*, *tasks=0*, *vrblvl=0*)

Runs one step in the cascade homotopy defined by the embedding of Laurent system in *embpols*, starting at the solutions in *nonsols*, removing the last hyperplane from *embpols* at dimension *dim*. The tolerance *tol* is used to split filter the solutions. Computations happen in quad double precision. If *vrblvl* > 0, then extra output is written.

cascades.**quad_double_laurent_cascade_step**(*dim*, *embsys*, *esols*, *tasks=0*, *vrblvl=0*)

Given in *embsys* an embedded Laurent polynomial system and solutions with nonzero slack variables in *esols*, does one step in the homotopy cascade, with quad double precision arithmetic. The dimension of the solution set represented by *embsys* and *esols* is the value of *dim*. The number of tasks in multithreaded path tracking is given by *tasks*. The default zero value of *tasks* indicates no multithreading. The list on return contains witness points on lower dimensional solution components.

cascades.**quad_double_laurent_top_cascade**(*nvr*, *dim*, *pols*, *tol=1e-06*, *tasks=0*, *vrblvl=0*)

Constructs an embedding of the Laurent polynomials in *pols*, with the number of variables in *pols* equal to *nvr*, where *dim* is the top dimension of the solution set. Applies the blackbox solver to the embedded system to compute in quad double precision the generic points and the nonsolutions for use in the cascade. Returns a tuple with three items:

1. the embedded system,
2. the solutions with zero last coordinate w.r.t. *tol*,
3. the solutions with nonzero last coordinate w.r.t. *tol*.

The three parameters on input are

1. *tol* is the tolerance to decide whether a number is zero or not, used to split the solution list of the embedded system;
2. *tasks* is the number of tasks, 0 if no multitasking,

3. if *vrblvl* > 0, then extra output is written.

cascades.**quad_double_top_cascade**(*nvr*, *dim*, *pols*, *tol=1e-06*, *tasks=0*, *vrblvl=0*)

Constructs an embedding of the polynomials in *pols*, with the number of variables in *pols* equal to *nvr*, where *dim* is the top dimension of the solution set. Applies the blackbox solver to the embedded system to compute in quad double precision the generic points and the nonsolutions for use in the cascade. Returns a tuple with three items:

1. the embedded system,
2. the solutions with zero last coordinate w.r.t. *tol*,
3. the solutions with nonzero last coordinate w.r.t. *tol*.

The three parameters on input are

1. *tol* is the tolerance to decide whether a number is zero or not, used to split the solution list of the embedded system;
2. *tasks* is the number of tasks, 0 if no multitasking,
3. if *vrblvl* > 0, then extra output is written.

cascades.**set_double_cascade_homotopy**(*vrblvl=0*)

Defines the cascade homotopy in double precision, called as a function in the double cascade step. The verbose level is given by vrblvl.

cascades.**set_double_double_cascade_homotopy**(*vrblvl=0*)

Defines the cascade homotopy in double double precision, called as a function in the double double cascade step. The verbose level is given by vrblvl.

cascades.**set_double_double_laurent_cascade_homotopy**(*vrblvl=0*)

Defines the cascade homotopy for Laurent systems in double double precision, called as a function in the double double cascade step. The verbose level is given by vrblvl.

cascades.**set_double_laurent_cascade_homotopy**(*vrblvl=0*)

Defines the cascade homotopy for Laurent systems in double precision, called as a function in the double cascade step. The verbose level is given by vrblvl.

cascades.**set_quad_double_cascade_homotopy**(*vrblvl=0*)

Defines the cascade homotopy in quad double precision, called as a function in the quad double cascade step. The verbose level is given by vrblvl.

cascades.**set_quad_double_laurent_cascade_homotopy**(*vrblvl=0*)

Defines the cascade homotopy for Laurent systems in quad double precision, called as a function in the quad double cascade step. The verbose level is given by vrblvl.

cascades.**split_filter**(*sols*, *dim*, *tol*, *vrblvl=0*)

Given in *sols* is a list of solutions of dimension *dim*, which contain a variable with name 'zz' + str(*dim*), which is the name of the last slack variable. The tolerance *tol* is used to split the list of solution in two. On return is a tuple of two lists of solutions (possibly empty). The first list of solutions has the last slack variable equal to zero (with respect to the tolerance *tol) and the last slack variable of each solution in the second list has a magnitude larger than *tol*. If *vrblvl* is nonzero, then the length of each solution list is printed.

cascades.**test_double_cascade**(*vrblvl=0*)

Tests one cascade step in double precision. In the top embedding we first find the 2-dimensional solution set x = 1. In the cascade step we compute the candidate witness points on the twisted cubic.

cascades.**test_double_double_cascade**(*vrblvl=0*)

Tests one cascade step in double double precision. In the top embedding we first find the 2-dimensional solution set x = 1. In the cascade step we compute the candidate witness points on the twisted cubic.

cascades.**test_double_double_laurent_cascade**(*vrblvl=0*)

Tests one cascade step on a Laurent system, in double double precision. In the top embedding we first find the 2-dimensional solution set x^-1 = 1. In the cascade step we compute the candidate witness points on the twisted cubic.

cascades.**test_double_laurent_cascade**(*vrblvl=0*)

Tests one cascade step on a Laurent system, in double precision. In the top embedding we first find the 2-dimensional solution set x^-1 = 1. In the cascade step we compute the candidate witness points on the twisted cubic.

cascades.**test_quad_double_cascade**(*vrblvl=0*)

Tests one cascade step in quad double precision. In the top embedding we first find the 2-dimensional solution set x = 1. In the cascade step we compute the candidate witness points on the twisted cubic.

cascades.**test_quad_double_laurent_cascade**(*vrblvl=0*)

Tests one cascade step on a Laurent system, in quad double precision. In the top embedding we first find the 2-dimensional solution set x^-1 = 1. In the cascade step we compute the candidate witness points on the twisted cubic.

4.5.3 functions in the module diagonal

Given two witness sets for two pure dimensional solution sets, a diagonal homotopy computes a sets of witness sets for all components of the intersection of the two pure dimensional solution sets.

diagonal.**bottom_diagonal_dimension**(*kdm*, *dim1*, *dim2*)

Returns the lowest dimension of the solution when intersecting two systems of dimensions dim1 and dim2 in dimension kdm.

diagonal.**diagonal_symbols_doubler**(*nbr*, *dim*, *symbols*, *vrblvl=0*)

Doubles the number of symbols in the symbol table to enable the writing of the target system to string properly when starting the cascade of a diagonal homotopy in extrinsic coordinates. On input are nbr, the ambient dimension = #variables before the embedding, dim is the number of slack variables, or the dimension of the first set, and in symbols are the symbols for the first witness set. This function takes the symbols in s and combines those symbols with those in the current symbol table for the second witness set stored in the standard systems container. On return, the symbol table contains then all symbols to write the top system in the cascade to start the diagonal homotopy.

diagonal.**double_collapse_diagonal**(*ksl*, *dim*, *vrblvl=0*)

Eliminates the extrinsic diagonal for the system and solutions in double precision. The number of slack variables in the current embedding is ksl, and dim is the number of slack variables to add to the final embedding.

diagonal.**double_diagonal_cascade_solutions**(*dim1*, *dim2*, *vrblvl=0*)

Defines the start solutions in the cascade to start the diagonal homotopy to intersect a set of dimension *dim1* with another set of dimension *dim2*, in double precision. For this to work, double_diagonal_homotopy must have been executed successfully.

diagonal.**double_diagonal_homotopy**(*dim1*, *sys1*, *esols1*, *dim2*, *sys2*, *esols2*, *vrblvl=0*)

Defines a diagonal homotopy to intersect the witness sets defined by (*sys1*, *esols1*) and (*sys2*, *esols2*), respectively of dimensions *dim1* and *dim2*. The systems *sys1* and *sys2* are assumed to be square and with as many slack variables as the dimension of the solution sets. The data is stored in double precision.

diagonal.**double_diagonal_solve**(*dim*, *dm1*, *sys1*, *sols1*, *dm2*, *sys2*, *sols2*, *tasks=0*, *vrblvl=0*)

Runs the diagonal homotopies in double precision to intersect two witness sets stored in (*sys1*, *sols1*) and (*sys2*, *sols2*), of respective dimensions *dm1* and *dm2*. The ambient dimension equals *dim*. Multitasking is available, and is activated by the *tasks* parameter. Returns the last system in the cascade and its solutions.

diagonal.**double_double_collapse_diagonal**(*ksl*, *dim*, *vrblvl=0*)

Eliminates the extrinsic diagonal for the system and solutions in double double precision. The number of slack variables in the current embedding is ksl, and dim is the number of slack variables to add to the final embedding.

diagonal.**double_double_diagonal_cascade_solutions**(*dim1*, *dim2*, *vrblvl=0*)

Defines the start solutions in the cascade to start the diagonal homotopy to intersect a set of dimension *dim1* with another set of dimension *dim2*, in double double precision. For this to work, double_double_diagonal_homotopy must have been executed successfully.

diagonal.**double_double_diagonal_homotopy**(*dim1*, *sys1*, *esols1*, *dim2*, *sys2*, *esols2*, *vrblvl=0*)

Defines a diagonal homotopy to intersect the witness sets defined by (*sys1*, *esols1*) and (*sys2*, *esols2*), respectively of dimensions *dim1* and *dim2*. The systems *sys1* and *sys2* are assumed to be square and with as many slack variables as the dimension of the solution sets. The data is stored in double double precision.

diagonal.**double_double_diagonal_solve**(*dim*, *dm1*, *sys1*, *sols1*, *dm2*, *sys2*, *sols2*, *tasks=0*, *vrblvl=0*)

Runs the diagonal homotopies in double double precision to intersect two witness sets stored in (*sys1*, *sols1*) and (*sys2*, *sols2*), of respective dimensions *dm1* and *dm2*. The ambient dimension equals *dim*. Multitasking is available, and is activated by the *tasks* parameter. Returns the last system in the cascade and its solutions.

diagonal.**double_double_start_diagonal_cascade**(*gamma=0*, *tasks=0*, *vrblvl=0*)

Does the path tracking to start a diagonal cascade in double double precision. For this to work, the double_double_diagonal_homotopy and double_double_diagonal_cascade_solutions must be executed successfully. If *gamma* equals 0 on input, then a random gamma constant is generated, otherwise, the given complex gamma will be used in the homotopy. Multitasking is available, and activated by the *tasks* parameter. Returns the target (system and its corresponding) solutions.

diagonal.**double_start_diagonal_cascade**(*gamma=0*, *tasks=0*, *vrblvl=0*)

Does the path tracking to start a diagonal cascade in double precision. For this to work, the functions double_diagonal_homotopy and double_diagonal_cascade_solutions must be executed successfully. If *gamma* equals 0 on input, then a random gamma constant is generated, otherwise, the given complex gamma will be used in the homotopy. Multitasking is available, and activated by the *tasks* parameter. Returns the target (system and its corresponding) solutions.

diagonal.**extrinsic_top_diagonal_dimension**(*ambdim1*, *ambdim2*, *setdim1*, *setdim2*, *vrblvl=0*)

Returns the dimension of the start and target system to start the extrinsic cascade to intersect two witness sets, respectively of dimensions setdim1 and setdim2, with ambient dimensions respectively equal to ambdim1 and ambdim2.

diagonal.**main**()

Runs some tests.

diagonal.**quad_double_collapse_diagonal**(*ksl*, *dim*, *vrblvl=0*)

Eliminates the extrinsic diagonal for the system and solutions in quad double precision. The number of slack variables in the current embedding is ksl, and dim is the number of slack variables to add to the final embedding.

diagonal.**quad_double_diagonal_cascade_solutions**(*dim1*, *dim2*, *vrblvl=0*)

Defines the start solutions in the cascade to start the diagonal homotopy to intersect a set of dimension *dim1* with another set of dimension *dim2*, in quad double precision. For this to work, quad_double_diagonal_homotopy must have been executed successfully.

diagonal.**quad_double_diagonal_homotopy**(*dim1*, *sys1*, *esols1*, *dim2*, *sys2*, *esols2*, *vrblvl=0*)

Defines a diagonal homotopy to intersect the witness sets defined by (*sys1*, *esols1*) and (*sys2*, *esols2*), respectively of dimensions *dim1* and *dim2*. The systems *sys1* and *sys2* are assumed to be square and with as many slack variables as the dimension of the solution sets. The data is stored in quad double precision.

diagonal.**quad_double_diagonal_solve**(*dim*, *dm1*, *sys1*, *sols1*, *dm2*, *sys2*, *sols2*, *tasks=0*, *vrblvl=0*)

Runs the diagonal homotopies in quad double precision to intersect two witness sets stored in (*sys1*, *sols1*) and (*sys2*, *sols2*), of respective dimensions *dm1* and *dm2*. The ambient dimension equals *dim*. Multitasking is available, and is activated by the *tasks* parameter. Returns the last system in the cascade and its solutions.

diagonal.**quad_double_start_diagonal_cascade**(*gamma=0*, *tasks=0*, *vrblvl=0*)

Does the path tracking to start a diagonal cascade in quad double precision. For this to work, the quad_double_diagonal_homotopy and quad_double_diagonal_cascade_solutions must be executed successfully. If *gamma* equals 0 on input, then a random gamma constant is generated, otherwise, the given complex gamma will be used in the homotopy. Multitasking is available, and activated by the *tasks* parameter. Returns the target (system and its corresponding) solutions.

diagonal.**set_double_diagonal_homotopy**(*dim1*, *dim2*, *vrblvl=0*)

Defines a diagonal homotopy to intersect two solution sets of dimensions dim1 and dim2 respectively, where dim1 >= dim2. The systems that define the witness sets must have been set already in double precision before the call to this function.

diagonal.**set_double_double_diagonal_homotopy**(*dim1*, *dim2*, *vrblvl=0*)

Defines a diagonal homotopy to intersect two solution sets of dimensions dim1 and dim2 respectively, where dim1 >= dim2. The systems that define the witness sets must have been set already in double double precision before the call to this function.

diagonal.**set_quad_double_diagonal_homotopy**(*dim1*, *dim2*, *vrblvl=0*)

Defines a diagonal homotopy to intersect two solution sets of dimensions dim1 and dim2 respectively, where dim1 >= dim2. The systems that define the witness sets must have been set already in quad double precision before the call to this function.

diagonal.**test_double_diagonal_homotopy**(*vrblvl=0*)

Tests the diagonal homotopy in double precision.

diagonal.**test_double_double_diagonal_homotopy**(*vrblvl=0*)

Tests the diagonal homotopy in double double precision.

diagonal.**test_double_double_hypersurface_intersection**(*vrblvl=0*)

Tests the intersection of a cylinder and a sphere, in double double precision.

diagonal.**test_double_hypersurface_intersection**(*vrblvl=0*)

Tests the intersection of a cylinder and a sphere, in double precision.

diagonal.**test_quad_double_diagonal_homotopy**(*vrblvl=0*)

Tests the diagonal homotopy in quad double precision.

diagonal.**test_quad_double_hypersurface_intersection**(*vrblvl=0*)

Tests the intersection of a cylinder and a sphere, in quad double precision.

diagonal.**top_diagonal_dimension**(*kdm*, *dim1*, *dim2*)

Returns the number of slack variables at the top in the cascade of diagonal homotopies to intersect two sets of dimension *dim1* and *dim2*, where *dim1* >= *dim2* and *kdm* is the dimension before the embedding. Typically, *kdm* is the number of equations in the first witness set minus *dim1*.

4.5.4 functions in the module factor

Given a witness set representation of a pure dimensional solution set, the functions in this module separate the generic points in the witness set according to the irreducible components of the solution set.

factor.**copy_double_double_laurent_witness_set**(*vrblvl=0*)

Copies the witness set from the sampler to the laurent system set in double double precision.

factor.**copy_double_double_witness_set**(*vrblvl=0*)

Copies the witness set from the sampler to the system set in double double precision.

factor.**copy_double_laurent_witness_set**(*vrblvl=0*)

Copies the witness set from the sampler to the laurent system set in double precision.

factor.**copy_double_witness_set**(*vrblvl=0*)

Copies the witness set from the sampler to the system set in double precision.

factor.**copy_quad_double_laurent_witness_set**(*vrblvl=0*)

Copies the witness set from the sampler to the laurent system set in quad double precision.

factor.**copy_quad_double_witness_set**(*vrblvl=0*)

Copies the witness set from the sampler to the system set in quad double precision.

factor.**double_assign_labels**(*nvr*, *vrblvl=0*)

Assigns a unique label to the multiplicity field for each solution set in double precision.

factor.**double_decomposition**(*deg*, *vrblvl=0*)

Returns the decomposition as a list of labels of witness points on the components, computed in double precision.

factor.**double_double_assign_labels**(*nvr*, *vrblvl=0*)

Assigns a unique label to the multiplicity field for each solution set in double double precision.

factor.**double_double_decomposition**(*deg*, *vrblvl=0*)

Returns the decomposition as a list of labels of witness points on the components, computed in double double precision.

factor.**double_double_factor_count**(*vrblvl=0*)

Returns the number of factors computed in double double precision.

factor.**double_double_loop_permutation**(*deg*, *vrblvl=0*)

Returns the permutation using the solution most recently computed, for a set of degree deg, after a loop in double double precision.

factor.**double_double_monodromy_breakup**(*embsys*, *esols*, *dim*, *islaurent=False*, *verbose=False*, *nbloops=20*, *vrblvl=0*)

Applies the monodromy breakup algorithm in double double precision to factor the *dim*-dimensional algebraic set represented by the embedded system *embsys* and its solutions *esols*. If the embedded polynomial system is a laurent system, then *islaurent* must be True. If *verbose* is False, then no output is written. The value of *nbloops* equals the maximum number of loops.

factor.**double_double_trace_grid_diagnostics**(*vrblvl=0*)

Returns the maximal error on the samples in the trace grid and the minimal distance between the samples in the trace grid, computed in double double precision.

factor.**double_double_trace_sum_difference**(*labels*, *vrblvl=0*)

Returns the difference between the actual sum at the samples defined by the labels to the generic points of a factor and the trace sum, in double double precision.

factor.**double_double_trace_test**(*vrblvl=0*)

Runs the trace test on the decompostion in double double precision, returns True if certified, otherwise returns False.

factor.**double_double_witness_points**(*idx*, *deg*, *vrblvl=0*)

Given an index idx of an irreducible component, computed in double double precision, returns the labels of the witness points that span the component. The input deg is the upper bound on the degree of a factor. The degree of the factor is the length of the returned list.

factor.**double_double_witness_sample**(*vrblvl=0*)

Computes a new witness set for a new set of slices, in double double precision.

factor.**double_double_witness_track**(*islaurent=False*, *vrblvl=0*)

Tracks as many paths as set in double double precision, as many as the size of the witness set. If the system is laurent, then islaurent must be True.

factor.**double_factor_count**(*vrblvl=0*)

Returns the number of factors computed in double precision.

factor.**double_loop_permutation**(*deg*, *vrblvl=0*)

Returns the permutation using the solution most recently computed, for a set of degree deg, after a loop in double precision.

factor.**double_monodromy_breakup**(*embsys*, *esols*, *dim*, *islaurent=False*, *verbose=False*, *nbloops=20*, *vrblvl=0*)

Applies the monodromy breakup algorithm in double precision to factor the *dim*-dimensional algebraic set represented by the embedded system *embsys* and its solutions *esols*. If the embedded polynomial system is a laurent system, then *islaurent* must be True. If *verbose* is False, then no output is written. The value of *nbloops* equals the maximum number of loops.

factor.**double_trace_grid_diagnostics**(*vrblvl=0*)

Returns the maximal error on the samples in the trace grid and the minimal distance between the samples in the trace grid, computed in double precision.

factor.**double_trace_sum_difference**(*labels*, *vrblvl=0*)

Returns the difference between the actual sum at the samples defined by the labels to the generic points of a factor and the trace sum, in double precision.

factor.**double_trace_test**(*vrblvl=0*)

Runs the trace test on the decompostion in double precision, returns True if certified, otherwise returns False.

factor.**double_witness_points**(*idx*, *deg*, *vrblvl=0*)

Given an index idx of an irreducible component, computed in double precision, returns the labels of the witness points that span the component. The input deg is the upper bound on the degree of a factor. The degree of the factor is the length of the returned list.

factor.**double_witness_sample**(*vrblvl=0*)

Computes a new witness set for a new set of slices, in double precision.

factor.**double_witness_track**(*islaurent=False*, *vrblvl=0*)

Tracks as many paths as set in double precision, as many as the size of the witness set. If the system is laurent, then islaurent must be True.

factor.**initialize_double_double_laurent_sampler**(*dim*, *vrblvl=0*)

Initializes the sampling machine with a witness set, for a laurent system set in double double precision.

factor.**initialize_double_double_monodromy**(*nbloops*, *deg*, *dim*, *vrblvl=0*)

Initializes to run nbloops monodromy loops to factor a positive dimension solution set of dimension dim and of degree deg in double double precision.

factor.**initialize_double_double_sampler**(*dim*, *vrblvl=0*)

Initializes the sampling machine with a witness set, set in double double precision.

factor.**initialize_double_laurent_sampler**(*dim*, *vrblvl=0*)

Initializes the sampling machine with a witness set, for a laurent system set in double precision.

factor.**initialize_double_monodromy**(*nbloops*, *deg*, *dim*, *vrblvl=0*)

Initializes to run nbloops monodromy loops to factor a positive dimension solution set of dimension dim and of degree deg in double precision.

factor.**initialize_double_sampler**(*dim*, *vrblvl=0*)

Initializes the sampling machine with a witness set, set in double precision.

factor.**initialize_quad_double_laurent_sampler**(*dim*, *vrblvl=0*)

Initializes the sampling machine with a witness set, for a laurent system set in quad double precision.

factor.**initialize_quad_double_monodromy**(*nbloops*, *deg*, *dim*, *vrblvl=0*)

Initializes to run nbloops monodromy loops to factor a positive dimension solution set of dimension dim and of degree deg in double double precision.

factor.**initialize_quad_double_sampler**(*dim*, *vrblvl=0*)

Initializes the sampling machine with a witness set, set in quad double precision.

factor.**main**()

Runs some tests.

factor.**new_double_double_slices**(*dim*, *nvr*, *vrblvl=0*)

Generates dim random hyperplanes in double double precision where nvr is the number of variables.

factor.**new_double_slices**(*dim*, *nvr*, *vrblvl=0*)

Generates dim random hyperplanes in double precision where nvr is the number of variables.

factor.**new_quad_double_slices**(*dim*, *nvr*, *vrblvl=0*)

Generates dim random hyperplanes in quad double precision where nvr is the number of variables.

factor.**preset_double_double_solutions**(*vrblvl=0*)

Initializes the start solutions for monodromy loops to the solutions set in double double precision.

factor.**preset_double_solutions**(*vrblvl=0*)

Initializes the start solutions for monodromy loops to the solutions set in double precision.

factor.**preset_quad_double_solutions**(*vrblvl=0*)

Initializes the start solutions for monodromy loops to the solutions set in quad double precision.

factor.**quad_double_assign_labels**(*nvr*, *vrblvl=0*)

Assigns a unique label to the multiplicity field for each solution set in quad double precision.

factor.**quad_double_decomposition**(*deg*, *vrblvl=0*)

Returns the decomposition as a list of labels of witness points on the components, computed in quad double precision.

factor.**quad_double_factor_count**(*vrblvl=0*)

Returns the number of factors computed in quad double precision.

factor.**quad_double_loop_permutation**(*deg*, *vrblvl=0*)

Returns the permutation using the solution most recently computed, for a set of degree deg, after a loop in quad double precision.

factor.**quad_double_monodromy_breakup**(*embsys*, *esols*, *dim*, *islaurent=False*, *verbose=False*, *nbloops=20*, *vrblvl=0*)

Applies the monodromy breakup algorithm in quad double precision to factor the *dim*-dimensional algebraic set represented by the embedded system *embsys* and its solutions *esols*. If the embedded polynomial system is a laurent system, then *islaurent* must be True. If *verbose* is False, then no output is written. The value of *nbloops* equals the maximum number of loops.

factor.**quad_double_trace_grid_diagnostics**(*vrblvl=0*)

Returns the maximal error on the samples in the trace grid and the minimal distance between the samples in the trace grid, computed in quad double precision.

factor.**quad_double_trace_sum_difference**(*labels*, *vrblvl=0*)

Returns the difference between the actual sum at the samples defined by the labels to the generic points of a factor and the trace sum, in quad double precision.

factor.**quad_double_trace_test**(*vrblvl=0*)

Runs the trace test on the decompostion in quad double precision, returns True if certified, otherwise returns False.

factor.**quad_double_witness_points**(*idx*, *deg*, *vrblvl=0*)

Given an index idx of an irreducible component, computed in quad double precision, returns the labels of the witness points that span the component. The input deg is the upper bound on the degree of a factor. The degree of the factor is the length of the returned list.

factor.**quad_double_witness_sample**(*vrblvl=0*)

Computes a new witness set for a new set of slices, in quad double precision.

factor.**quad_double_witness_track**(*islaurent=False*, *vrblvl=0*)

Tracks as many paths as set in quad double precision, as many as the size of the witness set. If the system is laurent, then islaurent must be True.

factor.**reset_double_double_solutions**(*vrblvl=0*)

Resets the start solutions for monodromy loops in double double precision to the solutions used to initialize the start solutions with.

factor.**reset_double_solutions**(*vrblvl=0*)

Resets the start solutions for monodromy loops in double precision to the solutions used to initialize the start solutions with.

factor.**reset_quad_double_solutions**(*vrblvl=0*)

Resets the start solutions for monodromy loops in quad double precision to the solutions used to initialize the start solutions with.

factor.**set_double_double_gammas**(*dim*, *vrblvl=0*)

Sets the gamma constants in double_double precision for the sampler in the monodromy loops. Generates as many random complex constants as dim.

factor.**set_double_double_slice**(*equ*, *idx*, *cff*, *vrblvl=0*)

Sets the coefficients of slicing equation with index equ at position idx to the value in double double precision: (cff[0], cff[1]) + (cff[2], cff[3])*complex(0, 1).

factor.**set_double_double_trace**(*first*, *vrblvl=0*)

Sets the constant coefficient of the first slice, for use in the linear trace test in double double precision. The integer first indicates if it is the first time or not.

factor.**set_double_double_verbose**(*verbose=True*, *vrblvl=0*)

Sets the state of the monodromy algorithm in double double precision to verbose if vrblvl > 0, otherwise it is set to remain mute.

factor.**set_double_gammas**(*dim*, *vrblvl=0*)

Sets the gamma constants in double precision for the sampler in the monodromy loops. Generates as many random complex constants as dim.

factor.**set_double_slice**(*equ*, *idx*, *cff*, *vrblvl=0*)

Sets the coefficients of slicing equation with index equ at position idx to the value in double precision: cff[0] + cff[1]*complex(0, 1).

factor.**set_double_trace**(*first*, *vrblvl=0*)

Sets the constant coefficient of the first slice, for use in the linear trace test in double precision. The integer first indicates if it is the first time or not.

factor.**set_double_verbose**(*verbose=True*, *vrblvl=0*)

Sets the state of the monodromy algorithm in double precision to verbose if vrblvl > 0, otherwise it is set to remain mute.

factor.**set_quad_double_gammas**(*dim*, *vrblvl=0*)

Sets the gamma constants in quad_double precision for the sampler in the monodromy loops. Generates as many random complex constants as dim.

factor.**set_quad_double_slice**(*equ*, *idx*, *cff*, *vrblvl=0*)

Sets the coefficients of slicing equation with index equ at position idx to the value in quad double precision: (cff[0], cff[1], cff[2], cff[3]) + (cff[4], cff[5], cff[6], cff[7])*complex(0, 1).

factor.**set_quad_double_trace**(*first*, *vrblvl=0*)

Sets the constant coefficient of the first slice, for use in the linear trace test in quad double precision. The integer first indicates if it is the first time or not.

factor.**set_quad_double_verbose**(*verbose=True*, *vrblvl=0*)

Sets the state of the monodromy algorithm in quad double precision to verbose if vrblvl > 0, otherwise it is set to remain mute.

factor.**swap_double_double_slices**(*vrblvl=0*)

Swaps the slices with new ones to turn back in one monodromy loop in double double precision.

factor.**swap_double_slices**(*vrblvl=0*)

Swaps the slices with new ones to turn back in one monodromy loop in double precision.

factor.**swap_quad_double_slices**(*vrblvl=0*)

Swaps the slices with new ones to turn back in one monodromy loop in quad double precision.

factor.**test_double_assign_labels**(*vrblvl=0*)

Tests the assigning of labels for solutions set in double precision.

factor.**test_double_double_assign_labels**(*vrblvl=0*)

Tests the assigning of labels for solutions set in double double precision.

factor.**test_double_double_monodromy**(*vrblvl=0*)

Tests the monodromy breakup in double double precision.

factor.**test_double_monodromy**(*vrblvl=0*)

Tests the monodromy breakup in double precision.

factor.**test_quad_double_assign_labels**(*vrblvl=0*)

Tests the assigning of labels for solutions set in quad double precision.

factor.**test_quad_double_monodromy**(*vrblvl=0*)

Tests the monodromy breakup in quad double precision.

factor.**update_double_decomposition**(*deg*, *perm*, *vrblvl=0*)

Updates the decomposition with a permutation of deg elements computed in double precision. Returns the tuple with the previous and the new number of factors.

factor.**update_double_double_decomposition**(*deg*, *perm*, *vrblvl=0*)

Updates the decomposition with a permutation of deg elements computed in double double precision. Returns the tuple with the previous and the new number of factors.

factor.**update_quad_double_decomposition**(*deg*, *perm*, *vrblvl=0*)

Updates the decomposition with a permutation of deg elements computed in quad double precision. Returns the tuple with the previous and the new number of factors.

factor.**write_factorization**(*deco*)

Writes the decomposition in deco.

4.5.5 functions in the module decomposition

Exports functions to compute a numerical irreducible decomposition.

decomposition.**copy_double_double_laurent_witset**(*dim*, *vrblvl=0*)

Copies the witness set at dimension dim to the Laurent polynomials and the solutions in double double precision.

decomposition.**copy_double_double_witset**(*dim*, *vrblvl=0*)

Copies the witness set at dimension dim to the polynomials and the solutions in double double precision.

decomposition.**copy_double_laurent_witset**(*dim*, *vrblvl=0*)

Copies the witness set at dimension dim to the Laurent polynomials and the solutions in double precision.

decomposition.**copy_double_witset**(*dim*, *vrblvl=0*)

Copies the witness set at dimension dim to the polynomials and the solutions in double precision.

decomposition.**copy_quad_double_laurent_witset**(*dim*, *vrblvl=0*)

Copies the witness set at dimension dim to the Laurent polynomials and the solutions in quad double precision.

decomposition.**copy_quad_double_witset**(*dim*, *vrblvl=0*)

Copies the witness set at dimension dim to the polynomials and the solutions in quad double precision.

decomposition.**double_double_laurent_solve**(*pols*, *topdim=- 1*, *filtsols=True*, *factor=True*, *tasks=0*, *verbose=True*, *vrblvl=0*)

Runs the cascades of homotopies on the Laurent system in pols in double double precision. The default top dimension topdim is the number of variables in pols minus one. The other parameters are (1) filtsols, to filter the spurious solutions, (2) factor, to factor the positive dimensional components, (3) tasks, is the number of tasks (0 for no multithreading), (4) verbose, to write extra information during the decomposition. The list on return contains a witness set for every dimension. If factor, then the last element in the list on return is the string representation of the irreducible factors. The verbose level is given by vrblvl.

decomposition.**double_double_solve**(*pols*, *topdim=- 1*, *filtsols=True*, *factor=True*, *tasks=0*, *verbose=True*, *vrblvl=0*)

Runs the cascades of homotopies on the polynomial system in pols in double double precision. The default top dimension topdim is the number of variables in pols minus one. The other parameters are (1) filtsols, to filter the spurious solutions, (2) factor, to factor the positive dimensional components, (3) tasks, is the number of tasks (0 for no multithreading), (4) verbose, to write extra information during the decomposition. The list on return contains a witness set for every dimension. If factor, then the last element in the list on return is the string representation of the irreducible factors. The verbose level is given by vrblvl.

decomposition.**double_laurent_solve**(*pols*, *topdim=- 1*, *filtsols=True*, *factor=True*, *tasks=0*, *verbose=True*, *vrblvl=0*)

Runs the cascades of homotopies on the Laurent system in pols in double precision. The default top dimension topdim is the number of variables in pols minus one. The other parameters are (1) filtsols, to filter the spurious solutions, (2) factor, to factor the positive dimensional components, (3) tasks, is the number of tasks (0 for no multithreading), (4) verbose, to write extra information during the decomposition. The list on return contains a witness set for every dimension. If factor, then the last element in the list on return is the string representation of the irreducible factors. The verbose level is given by vrblvl.

decomposition.**double_solve**(*pols*, *topdim=- 1*, *filtsols=True*, *factor=True*, *tasks=0*, *verbose=True*, *vrblvl=0*)

Runs the cascades of homotopies on the polynomial system in pols in double precision. The default top dimension topdim is the number of variables in pols minus one. The other parameters are (1) filtsols, to filter the spurious solutions, (2) factor, to factor the positive dimensional components, (3) tasks, is the number of tasks (0 for no multithreading), (4) verbose, to write extra information during the decomposition. The list on return contains a witness set for every dimension. If factor, then the last element in the list on return is the string representation of the irreducible factors. The verbose level is given by vrblvl.

decomposition.**get_factors**(*size*, *vrblvl=0*)

Returns the string representation of the irreducible factors, given in size is the number of characters in the string.

decomposition.**main**()

Runs some tests. Because of the sensitivity on tolerances, the seed of the random number generators are set.

decomposition.**quad_double_laurent_solve**(*pols*, *topdim=- 1*, *filtsols=True*, *factor=True*, *tasks=0*, *verbose=True*, *vrblvl=0*)

Runs the cascades of homotopies on the Laurent system in pols in quad double precision. The default top dimension topdim is the number of variables in pols minus one. The other parameters are (1) filtsols, to filter the spurious solutions, (2) factor, to factor the positive dimensional components, (3) tasks, is the number of tasks (0 for no multithreading), (4) verbose, to write extra information during the decomposition. The list on return contains a witness set for every dimension. If factor, then the last element in the list on return is the string representation of the irreducible factors. The verbose level is given by vrblvl.

decomposition.**quad_double_solve**(*pols*, *topdim=- 1*, *filtsols=True*, *factor=True*, *tasks=0*, *verbose=True*, *vrblvl=0*)

Runs the cascades of homotopies on the polynomial system in pols in quad double precision. The default top dimension topdim is the number of variables in pols minus one. The other parameters are (1) filtsols, to filter the spurious solutions, (2) factor, to factor the positive dimensional components, (3) tasks, is the number of tasks (0 for no multithreading), (4) verbose, to write extra information during the decomposition. The list on return contains a witness set for every dimension. If factor, then the last element in the list on return is the string representation of the irreducible factors. The verbose level is given by vrblvl.

decomposition.**solve**(*pols*, *topdim=- 1*, *filtsols=True*, *factor=True*, *tasks=0*, *precision='d'*, *verbose=True*, *vrblvl=0*)

Computes an irreducible decomposition of the polynomials in pols. The default top dimension topdim is the number of variables in pols minus one. The other parameters are (1) filtsols, to filter the spurious solutions, (2) factor, to factor the positive dimensional components, (3) tasks, is the number of tasks (0 for no multithreading),

(4) is the precision, by default double, (5) verbose, to write extra information during the decomposition. The list on return contains a witness set for every dimension. If factor, then the last element in the list on return is the string representation of the irreducible factors. The verbose level is given by vrblvl.

decomposition.**test_double_double_laurent_solve**(*vrblvl=0*)

Runs a test on solving a Laurent system in double double precision.

decomposition.**test_double_double_solve**(*vrblvl=0*)

Runs a test on solving in double double precision.

decomposition.**test_double_laurent_solve**(*vrblvl=0*)

Runs a test on solving a Laurent system in double precision.

decomposition.**test_double_solve**(*vrblvl=0*)

Runs a test on solving in double precision.

decomposition.**test_quad_double_laurent_solve**(*vrblvl=0*)

Runs a test on solving a Laurent system in quad double precision.

decomposition.**test_quad_double_solve**(*vrblvl=0*)

Runs a test on solving in quad double precision.

decomposition.**test_solve**(*vrblvl=0*)

Runs a test on the wrapper solve.

decomposition.**write_decomposition**(*deco*, *vrblvl=0*)

Writes the decomposition in deco.

4.5.6 functions in the module binomials

A binomial system is a polynomial system with exactly two monomials with nonzero coefficient in every equation. Binomials systems can be solved much more efficiently than other polynomial systems. The positive dimensional solution sets of binomial systems are monomial maps.

binomials.**coefficients_double_map**(*dim*, *idx*, *nvr*, *vrblvl=0*)

Returns the complex coefficients of a map of dimension dim, of index idx, in nvr variables, in double precision.

binomials.**degree_double_map**(*dim*, *idx*, *vrblvl=0*)

Returns the degree of map with index idx (starting at one), of dimension dim, in double precision.

binomials.**double_map_top_dimension**(*vrblvl=0*)

Returns the top dimension of the solution maps, in double precision.

binomials.**double_solve**(*nvr*, *pols*, *puretopdim=False*, *vrblvl=0*)

If the polynomial system in pols in number of variables nvr is a binomial system, then the solution maps are returned. When the flag puretopdim is True, then the solver will only look for the solution sets of the expected top dimension, which will lead to a faster execution.

binomials.**double_string_map**(*dim*, *nvr*, *cff*, *exp*, *vrblvl=0*)

Returns the string representation of a map of dimension dim, in nvr variables, with double complex coefficients in cff, and exponents in exp.

binomials.**exponents_double_map**(*dim*, *idx*, *nvr*, *vrblvl=0*)

Returns the exponents of a map of dimension dim, of index idx, in nvr variables, in double precision.

binomials.**is_binomial_system**(*vrblvl=0*)

Returns True if the system stored as a Laurent system in double precision is a binomial system, return False otherwise. The verbose level is given by the value of vrblvl.

binomials.**main**()

Runs some tests.

binomials.**number_double_maps**(*dim*, *vrblvl=0*)

Returns the number of maps of dimension dim, in double precision.

binomials.**test_double_solve**(*vrblvl=0*)

Tests on solving a binomial system.

binomials.**test_is_binomial_system**(*vrblvl=0*)

Tests on the binomial system test.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

D

E

F

G

H

I

K

L

M

N

P

Q

R

S

T

U

V

W