## Contents

1 Getting Started ................................................................. 1
   1.1 what is phcpy? ............................................................. 1
   1.2 installing phcpy .......................................................... 2
   1.3 extending Sage with phcpy .............................................. 3
   1.4 project history ............................................................ 4
   1.5 references ................................................................. 5
   1.6 acknowledgments ........................................................ 5
   1.7 about this document ..................................................... 6

2 Tutorial ................................................................................. 7
   2.1 the circle problem of Apollonius ...................................... 7
      2.1.1 a general configuration ............................................. 9
      2.1.2 a special configuration ............................................ 11
      2.1.3 a perturbed configuration ........................................ 12
   2.2 the design of a 4-bar mechanism ...................................... 13
      2.2.1 a general configuration ............................................. 15
      2.2.2 visualization of a straight line design ....................... 16
   2.3 lines meeting four given lines ......................................... 17
      2.3.1 a general configuration ............................................. 19
      2.3.2 a real configuration ................................................ 20
   2.4 tangent lines to a circle .................................................. 21
      2.4.1 lines through the origin intersecting a circle ............... 22
      2.4.2 defining the equations for the singular locus ................ 23
      2.4.3 intersecting two algebraic sets .................................. 24
   2.5 all lines tangent to four spheres ...................................... 26
      2.5.1 four mutually touching spheres ................................ 27
      2.5.2 tangents lines of multiplicities two ......................... 28
      2.5.3 twelve real single tangent lines ............................... 28

3 User Manual ......................................................................... 31
   3.1 a blackbox solver for isolated solutions ............................ 32
      3.1.1 solving random trinomials and a particular trinomial system 32
      3.1.2 representations of solutions of polynomial systems .......... 34
      3.1.3 reproducible runs with fixed seeds ............................. 38
      3.1.4 root counting methods ............................................ 38
      3.1.5 Newton’s method and deflation .................................. 39
      3.1.6 the multiplicity of an isolated solution ....................... 40
4 Reference Manual

4.1 a blackbox solver for isolated solutions
| 4.1.1 | functions in the module solver | 89 |
| 4.1.2 | functions in the module solutions | 98 |
| 4.1.3 | the module polynomials | 101 |
| 4.2 | path trackers and sweep homotopies | 101 |
| 4.2.1 | functions in the module trackers | 101 |
| 4.2.2 | functions in the module sweepers | 106 |
| 4.2.3 | functions in the module tuning | 108 |
| 4.2.4 | functions in the module tropisms | 114 |
| 4.3 | positive dimensional solution sets | 116 |
| 4.3.1 | functions in the module sets | 117 |
| 4.3.2 | functions in the module cascades | 123 |
| 4.3.3 | functions in the module factor | 126 |
| 4.3.4 | functions in the module diagonal | 128 |
| 4.4 | some interesting families and examples | 130 |
| 4.4.1 | functions in the module examples | 130 |
| 4.4.2 | functions in the module families | 132 |
| 4.5 | numerical Schubert calculus | 134 |
| 4.5.1 | functions in the module schubert | 134 |
| 4.6 | Newton polytopes, monomial maps, and power series | 136 |
| 4.6.1 | functions in the module polytopes | 136 |
| 4.6.2 | functions in the module maps | 138 |
| 4.6.3 | functions in the module series | 139 |
| 4.6.4 | functions in the module curves | 142 |
| 4.7 | a graphical user interface | 149 |
| 4.7.1 | functions in the module dashboard | 149 |
| 4.8 | the module phcpy.phcpy2c3 | 150 |
| 4.8.1 | functions in the module interface | 150 |
| 4.8.2 | functions in the module phcpy2c3 | 154 |

5 Indices and tables 213

Python Module Index 215

Index 217
CHAPTER 1

Getting Started

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

This work is licensed under a Creative Commons Attribution-Share Alike 3.0 License.

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. See the References section for pointers to the literature.

While PHCpack has been under development for over twenty years, phcpy is still working through its proof-of-concept stage. In its present state, working with phcpy will require persistence and plenty of patience.

1.1 what is phcpy?

The main executable phe (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 the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

One application of phcpy is to run regression tests. The Python interface phcpy to PHCpack is a programmer’s interface. The long-term goal is to make PHCpack more versatile, at least for those programmers familiar with the Python scripting language.
1.2 installing phcpy

The source for PHCpack can be downloaded from <http://www.math.uic.edu/~jan/download.html> and is also available at <https://github.com/janverschelde/PHCpack>. For the installation from source, the gnu-ada compiler (available for free at <http://libre.adacore.com/> is needed. Also your Python interpreter must most likely have been built with gcc. In addition to the Python interpreter, the file Python.h of the developer’s libraries will need to exist on your system. Otherwise, PHCpack and phcpy are self contained and do not require the installation of other software.

Up to version 0.3.7, phcpy was written with versions 2.6 and 2.7 of Python. Version 0.3.8 of phcpy was ported to Python 3.5, using a modified C interface phcpy2c3.c and the corresponding shared object phcpy2c3.so.

The code runs on Red Hat Linux 64-bit, Ubuntu Linux, and on Mac OS X. There is no support for Windows. Below is a step-by-step installation procedure.

0. The gnu-ada compiler must be installed to compile the shared object file. Although several Linux distributions come with gcc that have Ada enabled, check whether gnatmake is in your execution path. In a terminal window, at the command prompt, type which gnatmake. If the system answers with the location of gnatmake, then the gnu-ada compiler is installed on your system.

If you have multiple versions of gcc installed on your system, then the binaries of the gnu-ada compiler should appear first in your execution path. Typing gcc -v should show for GNAT GPL in the reply, or most recently (in 2018): GNAT Community 2018.

If both which gnatmake and gcc -v gave satisfactory replies, then you can proceed to step 2 and skip the installation of the gnu-ada compiler.

1. By default one needs to have superuser privileges to install the gnu-ada compiler at standard locations, but otherwise it is not hard to install the gnu-ada compiler in your own directory.

Following the instructions of the gnu-ada compiler, the location with the binaries must be added in front of the execution path. You may have to edit .bashrc (for the Bourne shell) or .cshrc (for the C shell).


On Linux systems, make sure to compile and install the library with the option -fPIC. When configuring, run configure as ./configure CXX=/usr/bin/g++ CXXFLAGS='-fPIC -O3' to set the flags of the c++ compiler.

If you rather would not (or cannot) install QDlib, then it is possible to compile the library in your home directory. All you need is to be aware of the location of the header files for the include statement and you need the qdlib.a file for linking. For an example, consider the makefile for Windows computers. The makefile_windows builds phc.exe with the QD library compiled in a home directory, not installed on the system.

3. The source code directory of PHCpack contains the directory Objects, a directory with file makefile in it. Depending on whether you are on a Unix-like system or on a mac, you will have to edit the makefile so the MAKEFILE variable either refers to makefile_unix or to makefile_mac. Once the makefile is adjusted you could type, just as a test, make phc to compile the main executable program. Note that for phcpy, you will not need this executable.

4. To make the shared object file, your python system needs to have been installed with the development version, that is: the file Python.h must be available on your disk. Often, following binary installations of the Python interpreter, this Python.h might be absent.

If packaged distributions for the development version of Python fail, then you may have to download the source code from <http://www.python.org>, configure, compile, and install the Python system. An additional benefit of such a Python installation is that then the Python interpreter could be built with the gnu-ada compiler, so both the scripting environment as the compiled code are using the same version of gcc.
5. Once you have located the file `Python.h` on your system, you most likely will have to adjust the definitions in the files `makefile_unix` or `makefile_mac`. Assign to the variables `PYTHON` and `PYTHON3` the directories where `Python.h` is.

6. In the directory `Objects` of the PHCpack source distribution, type `make phcpy2c2.so` to make the shared object file for python2, or type `make phcpy2c3.so` for the python3 version of phcpy. If all goes well, the shared object `phcpy2c2.so` can then be found in `Python/PHCpy2/phcpy` or `phcpy2c3.so` is then in `Python/PHCpy3/phcpy`.

If you run the Python interpreter in a terminal window in the directory `phcpy` of the source code distribution, then the `import phcpy` should already work.

7. To extend your python2 installation with phcpy, go to the `Python/PHCpy2` directory and run `python2 setup.py install` as superuser or as sudoer. For python3, go to `PHCpy/phcpy3` and run `python3 setup.py install` as superuser or as sudoer.

The documentation is typeset with Sphinx. Sphinx depends on the default version of Python on your system. If phcpy is installed with a different version of Python than the version Sphinx depends on, then this may cause problems to typeset the documentation.

### 1.3 extending Sage with phcpy

The current version of Sage uses python2. So the instructions to extend Sage with phcpy work only with the Python2 version of phcpy.

If you have installed Sage from source on your computer, then this installation comes with its own python libraries and interpreter. Then it is not too much work any more (in comparison to the steps in last section) to extend the python interpreter of Sage with phcpy.

On Linux systems, locate the python interpreter of Sage. Most likely this python is `/local/bin` of in the downloaded directory. Use the absolute path name for the location of the Sage python interpreter and navigate to the `Python/PHCpy2` directory which contains the `setup.py` for phcpy. Once in `Python/PHCpy2`, type `python setup.py install` at the command prompt. This does not require superuser access, but you must execute this setup with the same account you used to install Sage with.

We check the installation at the command prompt, as shown in Fig. 1.1.

```
$ sage

SageMath version 7.4, Release Date: 2016-10-18
Type "notebook()" for the browser-based notebook interface.
Type "help()" for help.

sage: import phcpy
PHCv2.4.33 released 2017-01-05 works!
sage:
```

Fig. 1.1: Importing phcpy in a Sage terminal session.

On Mac OS X, extending Sage with phcpy requires a bit more work as the `phcpy2c2.so` must be compiled with the Python library that comes with the Sage installation. For this, the `makefile_mac` must be modified with the correct definition for the location of the Python library of Sage, as defined by `SAGEPYTHONLIB`. With this definition, do `make sage_phcpy2c2.so` and then move this file under the name `phcpy2c2.so` to the directory `/Python/PHCpy2/phcpy`. The installation is then similar as for Linux, type `python setup.py install` at...
the command prompt in the directory where setup.py exists and for python using the absolute file name of the executable, e.g., type /Users/jan/Downloads/sage-7.2/local/bin/python setup.py install.

Importing phcpy apparently changes the configuration of the signal handlers which may lead Sage 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 Sage. With the init_cysignals(), the PariError exception is handled and the user can continue the Sage 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.


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. The 1.x.y releases series will continue to develop phcpy with an object-oriented interface to the functionality of PHCpack.
1.5 references


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.
1.7 about this document

This document arose as an exercise in exploring restructured text and Sphinx. All good software documents contain the following four items: an installation guide, a getting started, a tutorial, and a reference manual. This document combines all four.
The purpose of this chapter is to introduce phcpy via some use cases.

In all cases, there are three stages:

1. Given a problem, formulate the polynomial equations which describe the problem. This formulation results in a list of string representations for polynomials in several variables. This list is the input for the second stage.

2. If the problem leads to a polynomial system for which only the isolated solution are of interest, then the blackbox solver as in the solve method of the solver method will do. Otherwise, for positive dimensional solution sets, cascades of homotopies and diagonal homotopies are needed.

3. The solvers return their results as string representations for solutions. To process the solutions we convert the string representations into Python dictionaries.

In the first use cases, plots of the solutions are made with matplotlib. To formulate the polynomial equations we may use sympy, as illustrated in the design of 4-bar mechanisms. In the problem of the four lines, the intersection conditions are verified with numpy.

The python interpreter in Sage can be extended to include phcpy. In the problem of all lines tangent to four given lines, the polynomial system is formulated with Sage and the visualization of the solutions is also done within Sage.

In all use cases, we distinguish between general instances of a problem (where the numbers for the parameters are chosen at random), and specific instances of the problem. For these specific instances, singular solutions are likely to occur.

2.1 the circle problem of Apollonius

The circle problem of Apollonius asks to find all circles tangent to three given circles. In Fig. 2.1, the input circles are shown as blue disks and the eight circles tangent to the three input circles are displayed in red.

This problem can be reformulated into that of solving eight polynomial systems. The number eight equals the number of solutions to this problem. This chapter presents a use case for phcpy. The plots are generated with matplotlib.
Fig. 2.1: A general configuration of the circle problem of Apollonius.
2.1.1 a general configuration

Without loss of generality, we may fix the first given circle on input as the unit circle, centered at the origin and with radius one. We may fix the coordinate system so that the center of the second given circle on input lies on the first coordinate axis. Then we have two parameters for the second circle: its $x$-coordinate $b_x$ of the center and its radius $r_2$. The remaining three parameters of this problem are the two coordinates of the third circle on input, $c_x$ and $c_y$, and its radius $r_3$.

There are thus five parameters in the input of this problem:

1. $b_x$: the $x$-coordinate of the center of the second circle,
2. $r_2$: the radius of the second circle,
3. $c_x$: the $x$-coordinate of the center of the third circle,
4. $c_y$: the $y$-coordinate of the center of the third circle,
5. $r_3$: the radius of the third circle.

The conditions on the coordinates $x$ and $y$ and the radius $r$ of the circles tangent to the three given circles are expressed in the following three polynomial equations:

\[
\begin{align*}
    x^2 + y^2 - (r \pm 1)^2 &= 0 \\
    (x - b_x)^2 + y^2 - (r \pm r_2)^2 &= 0 \\
    (x - c_x)^2 + (y - c_y)^2 - (r \pm r_3)^2 &= 0
\end{align*}
\]

where the $\pm$ indicates that the distance of the center of the touching circle to a given circle is either the sum or the difference of the radii of the circles. Choosing one $+$ or one $-$ for the $\pm$ in each equation gives one polynomial system. Making all combinations of the $\pm$ gives eight polynomial systems to solve. Of course, only those real solutions with positive radius are valid solutions. Polynomials are represented as strings as in the following function:

```python
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,
    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+%s)^2;' % (c2x, r2) 
    e3m = '(x-%.15f)^2 + (y-%.15f)^2 - (r-%.15f)^2;' % (c3x, c3y, r3) 
    e3p = '(x-%.15f)^2 + (y-%.15f)^2 - (r+%s)^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]
```

2.1. the circle problem of Apollonius
The blackbox solver is a numerical solver. The input coefficients are cast into double floats with 15 decimal places of precision. The following function takes on input the list of polynomial systems and solves the systems. The valid solutions are extracted and returned.

```python
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.
    """
    from phcpy.solver import solve
    from phcpy.solutions import strsol2dict, is_real
    (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, verbose=False)
        if verbose:
            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 code for the function `solve4circles` has two important statements:

1. `sols = solve(eqs, verbose=False)` calls the blackbox solver; and
2. `soldic = strsol2dict(sol)` converts a solution string `sol` into the dictionary `soldic`.

The `solve` takes on input a list of strings. Each string represents a polynomial in several variables. What is returned by `solve` is a list of string representations of solutions. The function `strsol2dict` takes on input a string representation of a solution and returns a dictionary. The keys contain the names of the variables and the corresponding values are complex numbers, with the coordinates of the solution.

Solving then a general configuration of three circles with centers at \((0, 0)\), \((2, 0)\), \((1, 1)\), with corresponding radii 1, \(2/3\), and 1/3 happens by two function calls:

```python
syst = polynomials(2, 2.0/3, 1, 1, 1.0/3)
sols = solve4circles(syst)
```
2.1.2 a special configuration

A special configuration of the input circles is when they mutually touch each other, as shown in Fig. 2.2.

![Fig. 2.2: A special configuration of the circle problem of Apollonius.](image)

The five input parameters for this special configuration are listed in the tuple \((2, 1, 1, \sqrt{3}, 1)\). The radius of each circle equals one. The second circle has its center at \((2, 0)\) and the center for the third circle is at \((1, \sqrt{3})\).

In this special configuration we see only five red circles, three of which are equal to the given circles. The three given circles count as double solutions to the problem.

We can see the multiplicity of the solutions with the following session in an interactive Python shell:

```python
def polynomials(a, b, c, d, e):
    return [a - 1, b, c - 1, d, e]

def solve(p, verbose=False):
    if verbose:
        print(p)
    return [1.0, 1.0, 1.0]

def print_solution(t, m):
    print(f't : {t}  m : {m}')

from math import sqrt
h = sqrt(3)
p = polynomials(2, 1, 1, h, 1)
sols = solve(p[3], verbose=False)
print(sols[0])

from time import time
start = time()
print_solution(t, m)
end = time()
print(f'runtime: {end - start}')
```

In what is printed, we recognize the first input circle:

```
t : 1.00000000000000E+00  3.32383584047048E+00
m : 2
the solution for t :
```

(continues on next page)
Observe the \( m = 2 \) which indicates the multiplicity of the solution.

### 2.1.3 a perturbed configuration

To see why we have to count some solutions of the special configuration twice, consider Fig. 2.3.

![A perturbed configuration of the circle problem of Apollonius.](image)

The five input parameters for this perturbed problem are listed in the tuple \((2.05, 1, 1.025, \sqrt{3} + 0.025, 1)\). Each circle still has radius one, but the input circles no longer touch each other.

In an interactive Python shell, we may compute the solutions as follows:

```python
from math import sqrt
h = sqrt(3)
p = polynomials(2.05, 1, 1.025, h+0.025, 1)
sols = solve(p[3], verbose=False)
print(sols[0])
```
What is printed defines the large perturbation of the unit circle:

| t : 1.00000000000000E+00 | 5.00522372103966E+00 |
| m : 1 | |
| the solution for t : | |
| x : -3.0900834843067E-01 | -2.43642416382302E-114 |
| y : -1.98600887619915E-01 | -1.37048859215045E-114 |
| r : 1.36735854321414E+00 | 2.43642416382302E-114 |
| == err : 3.322E-16 == rco : 5.942E-03 == res : 1.388E-15 |

The script `appolonius.py` in the source code distribution in the examples folder of PHCpy2 or PHCpy3 is a self contained script which solves the three instances of the problem of Apollonius.

### 2.2 the design of a 4-bar mechanism

Given two pivot points and five precision points for the coupler, the design problem asks to determine the lengths of the bars that allow the coupler to pass through the given precision points.

This chapter presents a use case for phcpy. The equations are generated with sympy and the plots are made with matplotlib.

The system is taken from 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. In Fig. 2.4, the precision points are taken from Problem 7 in the paper.

The first point in Fig. 2.4, at the top leftmost corner shows the five precision points, labeled with the numbers 0, 1, 2, 3, and 4. The two white triangles in each plot represent the fixed pivots. The next five plots show one position of the 4-bar mechanism. Each position passes through one of the prescribed precision points. The rotation angles and the coordinates for x and y for the initial position are obtained as solutions of a polynomial system.

For the formulation of the equations we follow the notation of the problem statement in the second section of the paper by Morgan and Wampler. The first pivot point is fixed at the origin and the coordinates of the other fixed pivot point are in \(a = (a_1, a_2)\). The coordinates of the five precision points are denoted by \(d_0, d_1, d_2, d_3,\) and \(d_4\). All vectors are column vectors and the superscript \(T\) denotes the transpose. The planar rotation matrices are defined by

\[
R_j = \begin{bmatrix} c_j & -s_j \\ s_j & c_j \end{bmatrix}, \quad j = 1, 2, 3, 4
\]

where \(c_j\) and \(s_j\) are respectively the cosines and sines of the rotation angles. The first four equations express the relationship between cosines and sines in the identities

\[
c_j^2 + s_j^2 - 1 = 0, \quad j = 1, 2, 3, 4.
\]

The second group of equations involves the vector \(x = (x_1, x_2)\) of variables. The first bar in the mechanism is between the pivots. The line segment between the first pivot at \((0, 0)\) and \(x\) represents the second bar in the 4-bar mechanism.

\[
[d_j^T R_j - d_0^T] x + \frac{1}{2} [d_j^T d_j - d_0^T d_0] = 0, \quad j = 1, 2, 3, 4.
\]

The third bar in the mechanism is represented between \(x\) and \(y = (y_1, y_2)\) and the fourth bar connects \(y\) and the second pivot at \(a\). The third group of equations involving \(y\) is defined by

\[
[(d_j^T - a^T) R_j - (d_0^T - a^T)] y + \left[ \frac{1}{2} (d_j^T d_j - d_0^T d_0) - a^T (d_j - d_0) \right] = 0, \quad j = 1, 2, 3, 4.
\]

So we end up with a system of twelve equations in twelve unknowns: \(c_1, s_1, c_2, s_2, c_3, s_3, c_4, s_4, x_1, x_2, y_1, y_2\) and ten parameters, the coordinates of the precision points \(d_0, d_1, d_2, d_3,\) and \(d_4\). The coordinates of the second fixed pivot point \(a\) are typically set to be \((1, 0)\).
Fig. 2.4: A straight line design of a 4-bar mechanism.
2.2.1 a general configuration

For a general configuration, we generate 5 points, with coordinates uniformly distributed in the interval $[-1, +1]$. With sympy, the points are stored in object of the type `Matrix` for the computations in the formulation of the equations.

```python
from sympy.matrices import Matrix
from random import uniform as u

d0 = Matrix((2, 1), lambda i,j: u(-1,+1))
d1 = Matrix((2, 1), lambda i,j: u(-1,+1))
d2 = Matrix((2, 1), lambda i,j: u(-1,+1))
d3 = Matrix((2, 1), lambda i,j: u(-1,+1))
d4 = Matrix((2, 1), lambda i,j: u(-1,+1))
```

The rotation matrices involve cosines and sines of angles.

```python
c1, s1 = var('c1, s1')
c2, s2 = var('c2, s2')
c3, s3 = var('c3, s3')
c4, s4 = var('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]])
```

Then the first four equations reflect the identity $\cos^2(t) + \sin^2(t) = 1$ for any angle $t$.

```python
p1 = 'c1^2 + s1^2 - 1;

Two more additional unknowns are the end points of the first bar, which is connected to the origin.

```python
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)
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
```

For the equations on $x1$ and $x2$ we conver to strings:

```python
s1 = str(e1x[0]) + ‘;’
s2 = str(e2x[0]) + ‘;’
s3 = str(e3x[0]) + ‘;’
s4 = str(e4x[0]) + ‘;’
```

The third group of equations on $Y$ involve the pivot $a$.

```python
a = Matrix(([1], [0]))
y1, y2 = var('y1, y2')
Y = Matrix(([y1], [y2]))
cly = c1x - a.transpose()*(d1 - d0)
```

(continues on next page)
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

The string representations are defined as follows:

s5 = str(e1y[0]) + ';

s6 = str(e2y[0]) + ';

s7 = str(e3y[0]) + ';

s8 = str(e4y[0]) + ';

Then we have the polynomial system in the list:

equ = [p1, p2, p3, p4, s1, s2, s3, s4, s5, s6, s7, s8]
print('the polynomial system :')
for pol in equ:
    print(pol)

Then, at last, we run the blackbox solver:

from phcpy.solver import solve
sols = solve(equ)
print('the solutions :')
for sol in sols:
    print(sol)
print('computed', len(sols), 'solutions')

For any general choice of precision points, the number of solutions should always be the same, that is: 36.

### 2.2.2 visualization of a straight line design

Of special interest are those 4-bar mechanisms where the five precision points are on a line, as such mechanisms can be applied to translate circular into linear motion or otherwise.

The coordinates of the following five precision points are copied from Problem 7 of the paper by Morgan and Wampler:

```python
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]])
```

These are the coordinates shown in Fig. 2.4. There are 33 solutions to the polynomial system formulated in the same fashion as in the previous section. Of those 33 solutions, 15 are real. Only real solutions can lead to valid designs. Not every real solution leads to a valid design. One condition is that the four angles computed from the cosine and sine coordinates must be ordered, so that the precision points are reached the same order as they are listed in the input.
In Fig. 2.5, the coupler curve for the straight line mechanism is shown. This coupler curve is traced by the tip of the triangle moved by the 4-bar mechanism. The matplotlib code is available in the script fourbar.py in the examples folder of the source code for the Python 2 and Python 3 distributions.

![Coupler curve](image)

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

### 2.3 lines meeting four given lines

Consider as given four lines, our problem is to compute all lines which meet the four given lines in a point. In Fig. 2.6, the four given lines are shown in blue while the lines that meet those four lines are drawn in red.

A line in projective 3-space is represented by two points, stored in the columns of a 4-by-2 matrix. So the space we work in is the complex 4-space. For this problem we have a formal root count, named after Pieri.

```python
from phcpy.schubert import pieri_root_count
rc = pieri_root_count(2, 2, 0, verbose=False)
```

In 4-space, the dimension of the input planes equals two and also the dimension of the output planes is two. The value returned in `rc` is two for this problem.
Fig. 2.6: Two red lines meet four blue lines in a point. Their intersection points are marked by red disks.
2.3.1 a general configuration

In a general configuration, random number generators are applied to determine the points which span the input lines. The solving of a general configuration is encapsulated in the function `solve_general`.

```python
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.
    """
    from numpy import array
    from phcpy.schubert import random_complex_matrix
    from phcpy.schubert import run_pieri_homotopies
    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, verbose=False)
    inplanes = [array(plane) for plane in ranplanes]
    outplanes = [solution_plane(mdim+pdim, pdim, sol) for sol in sols]
    return (inplanes, outplanes, pols, sols)
```

The solutions returned by `run_pieri_homotopies` are converted into numpy matrices, as defined by the function `solution_plane`.

```python
def solution_plane(rows, cols, sol):
    """
    Returns a sympy array with as many rows
    as the value of rows and with as many columns
    as the value of columns, using the string
    representation of a solution in sol.
    """
    from numpy import zeros
    from phcpy.solutions import coordinates
    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 = (int(name[1]), int(name[2]))
        result[i-1][j-1] = value
    return result
```

For the verification of the intersection conditions, the matrices of the input planes are concatenated to the solution planes and the determinant of the concatenated matrix is computed.

```python
def verify_determinants(inps, sols, verbose=True):
    """
```
(continues on next page)
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.

```python
from numpy import matrix
from numpy.linalg import det
checksum = 0
for sol in sols:
    if verbose:
        print('checking solution
', 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
```

Then the main() function contains the following code.

```python
(inp, otp, pols, sols) = solve_general(mdim, pdim, deg)
print('The input planes :')
for plane in inp:
    print(plane)
print('The solution planes :')
for plane in otp:
    print(plane)
check = verify_determinants(inp, otp)
print('Sum of absolute values of determinants :', check)
```

The polynomial system in pols with corresponding solutions in sols can be used as start system to solve specific problems, as will be done in the next section.

### 2.3.2 a real configuration

The solution of a real instance takes on input the system and corresponding solutions of a general instance.

```python
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.
    """
    from phcpy.schubert import real_osculating_planes
    from phcpy.schubert import make_pieri_system
    from phcpy.trackers import track
    oscplanes = real_osculating_planes(mdim, pdim, 0)
    target = make_pieri_system(mdim, pdim, 0, oscplanes, False)
    rtsols = track(target, start, sols)
    inplanes = [array(plane) for plane in oscplanes]
    outplanes = [solution_plane(mdim+pdim, pdim, sol) for sol in rtsols]
    return (inplanes, outplanes, target, rtsols)
```
The code for the `main()` is similar as when calling `solve_general()`, as shown above at the end of the previous section.

The points which span the planes are in projective 3-space, represented by four coordinates. In projective space, the coordinates belong to equivalence classes and all nonzero multiples of the four coordinates represented the same point. To map the points in affine space, all coordinates are divided by the first coordinate. After this division, the first coordinate equals one and is omitted. This mapping is done by the function `input_generators`.

```python
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:])
```

The solutions of the Pieri homotopies are represented in a so-called localization pattern, where the second point has its first coordinate equal to zero. To map to affine 3-space, the second point is the sum of the two generators. The function `output_generators` below computes this mapping.

```python
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:])
```

The complete script is available in the directory `examples` of the source code for phcpy.

## 2.4 tangent lines to a circle

Given a fixed circle in the plane, compute the lines tangent to the circle and passing through the origin. In Fig. 2.7 we see a general line through the origin and two lines touching the circle.

The tangent lines are special: at the points where the lines touch the circle, we have a double solution, a solution of multiplicity two. One method is to consider the one parameter family of lines through the origin and intersect this family with the polynomials which express the singularity condition.
2.4.1 lines through the origin intersecting a circle

The polynomials which express all lines through the origin intersecting a fixed circle, fixed by its center and radius, are returned by a function.

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

There are two equations, one for the circle and one for the line. The variables are two coordinates \(x\), \(y\), and the slope \(s\). When given two equations in three variables we expect a one dimensional solution set. To represent this space curve, we intersect the curve with a general hyperplane and compute the points on the curve and on the hyperplane.

The code snippet below defines the problem for a circle centered at \((3, 2)\) with radius one. The `embed` function returns the original polynomials with one general hyperplane added and also one slack variable. The blackbox solver computes the generic points.

```python
pols = polynomials(3, 2, 1)
from phcpy.sets import embed
from phcpy.solver import solve
embpols = embed(3, 1, pols)
embsols = solve(embpols)
```

The tuple \((embpols, embsols)\) is a numerical representation for the set of all lines through the origin intersecting a fixed circle.

As a sanity check, consider a point on the set of all lines as in the left of Fig. 2.7. Such a point is for instance the line with slope one. The coordinates for the intersection points, as can be seen from Fig. 2.7 are \((2, 2)\) and \((3, 3)\). In the code below, the intersection point \((2, 2)\) is joined with the slope 1 in a solution string, called `point`.

```python
from phcpy.solutions import make_solution
point = make_solution(['x', 'y', 's'], [2, 2, 1])
ismb = is_member(embpols, embsols, 1, point)
```
The call to is_member returns a boolean value, so ismb should hold the value True for this point.

### 2.4.2 defining the equations for the singular locus

The two tangent lines to the circle are two special solutions. At any other line through the origin, the line intersects the circle at two distinct complex solutions, but at the tangent lines, the two intersection points collide into a double solution. At a double solution, the Jacobian matrix of the system no longer has full rank. Instead of using the determinant of the matrix of all first order partial derivatives, the equations we use express that there is a nonzero combination of the columns of the Jacobian matrix which yields the zero vector.

The equations for the singular locus are defined by the function jacobian. For the circle centered at \((3, 2)\), the polynomial equations are obtained as follows:

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

What is printed is

```plaintext
2*(x-3.000000000000000e+00)*L1 + 2*(y-2.000000000000000e+00)*L2;
-s*L1 + L2;
(-0.94944388496-0.313936791907*i) +(0.253472461117-0.967342602936*i)*L1 \
+(-0.209901989746-0.977722433234+i)*L2;
```

In the first equation we recognize the two partial derivatives of \((x-3)^2 + (y-2)^2\), multiplied with the multipliers \(L1\) and \(L2\). The second equation is derived from \(y - sx = 0\). If we have a nonzero combination of the columns of the Jacobian matrix which yields the zero vector, then any nonzero multiple of the multipliers also defines such a nonzero combination. The last equation is a linear equation in the multipliers only, requires that the multipliers are nonzero, and at the same time fixing one combination among all nonzero multiples.

The definition of the function jacobian depends on another function which returns a linear equation with random coefficients.

```python
def jacobian(a, b):
    ""
    For the circle centered at \((a, b)\),
    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]
```

To avoid badly scaled coefficients, the complex numbers are generated on the unit circle, but the function random_complex below.

```python
def random_complex():
    ""
    Returns a random complex number on the unit circle.
    ""
    from math import cos, sin, pi
    from random import uniform
    return uniform(cos(2*pi)) + uniform(sin(2*pi))*1j
```

(continues on next page)
The imaginary unit in Python is represented by \( j \) whereas for phcpy, the imaginary unit is represented by \( i \) and \( I \). Therefore, the function `random_hyperplane` replaces the \( j \) by \( i \).

```python
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 + ';'"""
```

The function \( \text{jacobian}(3, 2) \) returned three equations in the two coordinates \( x, y \), the slope \( s \), the multipliers \( L_1 \), and \( L_2 \); five variables in total. In five dimensional space, three equations define a two dimensional set.

For a numerical representation of this two dimensional set, two random linear equations are added with the `embed` function and the generic points are computed with the blackbox solver as done in the code snippet below.

```python
from phcpy.sets import embed
from phcpy.solver import solve
embpols = embed(5, 2, pols)
embsols = solve(embpols)
```

The number of generic points equals three.

### 2.4.3 intersecting two algebraic sets

We have two algebraic sets:

1. The set of all lines through the origin intersecting a fixed circle. The degree of this set is four.
2. The set of all intersection points of a line through the origin and a fixed circle where the Jacobian matrix is singular. The degree of this set is three.

Before we can intersect the two algebraic sets, we have to ensure that their ambient space is the same. The first set involves only the variables \( x, y \), and \( s \), but not the multiplier variables \( L_1 \) and \( L_2 \) which occur in the second algebraic set. Therefore, to each generic point in the first one dimensional set we add two values for \( L_1 \) and \( L_2 \) and two corresponding linear equations. So, the one dimensional set is upgraded to a three dimensional sets in the same five dimensional space in where the second two dimensional set lives. Because we can choose any values for \( L_1 \) and \( L_2 \) in this upgrade of the first set, the dimension of the first set increase from one to three.

Add two variable names \( L_1 \) and \( L_2 \), both with values one and two slack variables \( \text{zz2} \) and \( \text{zz3} \) with zero values is done by the function `extend_solutions`.

```python
def extend_solutions(sols):
    """
    To each solution in sols, adds \( L_1 \) and \( L_2 \) with values 1, and \( \text{zz2} \) and \( \text{zz3} \) with values zero.
    """
```

(continues on next page)
from phcpy.solutions import make_solution, coordinates
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

The function is called in the function `extend` which upgrades the first set from a one dimensional to a three dimensional set, raising its ambient space from a 3-space to the 5-space where the second set lives.

def extend(pols, sols):
    
    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)
return (extpols, extsols)

Note that the order of the equations is important. The linear equations that cut down the positive dimensional solutions to isolated points must occur at the end of the list of polynomials.

Also the order of the variables matters. To ensure that the names of the variables line up in the same order for both lists of polynomials, the first polynomial for both sets is prepended with the string x-x+y-y+s-s+L1-L1+L2-L2.

The relevant code snippet to intersect two sets with diagonal homotopies is shown below.

from phcpy.diagonal import diagonal_solver as diagsolve
result = diagsolve(dim, w1d, w1eqs, w1sols, w2d, w2eqs, w2sols)
(eqs, sols) = result

The polynomials and the corresponding generic points for the first set are in `w1eqs` and `w1sols` respectively, for the second set they are in `w2eqs` and `w2sols`. The dimensions of the two sets are in `w1d` and `w2d` (which respectively equal three and two) and the ambient dimension (five) is given in `dim`.

The number of solutions in list `sols` returned by the diagonal solver equals two, defining the two tangent lines shown at the right of Fig. 2.7.

The complete script which computes this use case is in the examples folder in the Python/PHCpy3 directory of the source code.

2.4. tangent lines to a circle
2.5 all lines tangent to four spheres

The problem is to find all lines that are tangent to four spheres. In Fig. 2.8, the special case is shown where the four spheres are mutually touching each other. In this case, a tangent line goes from one pair of touching spheres to another, opposite pair of touching spheres.

Solving the polynomial system associated with the configuration in Fig. 2.8 shows that each tangent line occurs with multiplicity four. Counting with multiplicities, the number of tangent lines thus equals twelve.

This problem was studied by I.G. Macdonald, J. Pach, and T. Theobald, who wrote the paper Common Tangents to Four Unit Balls in $\mathbb{R}^3$ published in Discrete and Computational Geometry 26(1): 1-17, 2001. Many instances of this problem appear in the book by Frank Sottile, entitled Real Solutions to Equations from Geometry published as

The tangent lines are represented in Pluecker coordinates, with a tangent $t$ and moment $m$ vector. For a point $p$ on the line, its cross product with the tangent vector equals the moment vector, or in equation form: $m = p \times t$. For a reference on this representation, consider the book Mechanics of robotic manipulation by Matthew T. Mason, MIT Press, 2001.

2.5.1 four mutually touching spheres

The centers of the four mutually touching spheres in Fig. 2.8 are $(+1, +1, +1)$, $(+1, -1, -1)$, $(-1, +1, -1)$, $(-1, -1, +1)$, and the radius is the same for all four spheres: $\sqrt{2}$.

The tangent lines are defined by a moment vector $m = (x_0, x_1, x_2)$ and a tangent vector $t = (x_3, x_4, x_5)$. The moment vector $m$ is perpendicular to the tangent vector $t$, which gives the first equation: $x_0 x_3 + x_1 x_4 + x_2 x_5 = 0$. The tangent vector is normalized: $||t||_2 = 1$, which gives the second equation $x_3^2 + x_4^2 + x_5^2 = 1$. For each center $c$ and radius $r$ of a sphere, the equation is

$$(m - c \times t) \cdot (m - c \times t) - r^2 = 0,$$

where $\times$ is the cross product and where $\cdot$ is the dot product. So we end up with a polynomial system of six equations in six unknowns.

The code in Sage to generate the polynomial system is below:

```python
x0, x1, x2 = var('x0, x1, x2')
t = (x0, x1, x2)
vt = vector(t)  # tangent vector
normt = vt.dot_product(vt) - 1
x3, x4, x5 = var('x3, x4, x5')
m = (x3, x4, x5)
vm = vector(m)  # moment vector
momt = vt.dot_product(vm)
seqs = [normt, momt]
for (ctr, rad) in zip(centers, radii):
    print 'the center :', ctr
    vc = vector(ctr)
    left = vm - vc.cross_product(vt)
equ = left.dot_product(left) - rad**2
    eqs.append(equ)
```

Then the input system for the blackbox solver of phcpy is the list of the string representations of the polynomials in `eqs`.

```python
polysys = []
for equ in eqs:
    pol = str(equ) + ';
    polysys.append(pol)
```

Calling the blackbox solver then happens as

```python
from phcpy.solver import solve
sols = solve(pols, verbose=False)
for sol in sols:
    print sol
```
and we see the multiplicity four solutions printed.

Lines are represented as \( \mathbf{m} = \mathbf{p} \times \mathbf{t} \), where \( \mathbf{p} \) is a point on the line. The solutions of the polynomial system give values for the components of the moment vector \( \mathbf{m} = (x_0, x_1, x_2) \) and the tangent vector \( \mathbf{t} = (x_3, x_4, x_5) \). To draw the line defined by \( \mathbf{m} \) and \( \mathbf{t} \) we need to compute the coordinates of \( \mathbf{p} = (p_1, p_2, p_3) \) which can be done via a simple cross product, because the tangent vector \( \mathbf{t} \) is normalized to one. The cross product \( \mathbf{p} = \mathbf{t} \times \mathbf{m} \) gives the coordinates of the point on the line closest to the origin.

### 2.5.2 tangents lines of multiplicities two

If the four spheres are centered at \((2, 2, 0), (2, 0, 2), (0, 2, 2), (0, 0, 0)\), and the radius of all four spheres is \(3/2\), then there are six lines tangents to all four spheres, which are to be counted each with multiplicity two, shown in Fig. 2.9.


The setup for the polynomial systems is identical to that of the previous section.

### 2.5.3 twelve real single tangent lines

A configuration with twelve real tangent lines of multiplicity one can be obtained by changing the radii in Fig. 2.8. Instead of taking \(\sqrt{2}\) as the value for each radius, the radius of each sphere is enlarged to \(\sqrt{2.01}\). This change is large enough for the quadruple tangent lines to split into single tangent lines and small enough for the single tangent lines to appears in clustered groups of four each, as shown in Fig. 2.10.

The script tangents4spheres.sage and the Sage notebook tangents4spheres.sws in the examples folder of the src/Python/PHCpy2 source distribution provide all details of the calculations.
Fig. 2.9: Six lines touching four spheres.
Fig. 2.10: Twelve single real tangent lines clustered in groups of four.
This chapter starts with a description of the blackbox solver, provided by the solver module. If solutions of a start system for a polynomial homotopy are available, then we may better call directly the path tracking routines to solve the target system in the homotopy. The path tracking functions are described in the second section.

The third section deals with the computation of positive dimensional solution sets. In a numerical irreducible decomposition of the solution set of a polynomial system, generic points are computed on all irreducible factors of the solution sets of all dimensions. In a witness set representation, the number of generic points in the witness set equals the degree of the pure dimensional solution set. Solution sets can be computed in a top down fashion with cascade homotopies or in a bottom up manner via diagonal homotopies.

The definitions of the polynomial systems which make interesting examples and families of problems are illustrated in section four.

Pieri homotopies and Littlewood-Richardson homotopies solve problems in enumerative geometry. The fifth section of this chapter is concerned with numerical Schubert calculus.

Every polynomial in several variables has a Newton polytope, spanned by the exponents of the monomials which occur with a nonzero coefficient. The mixed volume of a tuple of Newton polytopes gives an upper bound for the number of isolated solutions. Systems with exactly two monomials in every equation can be solved fast, via unimodular coordinate transformations. Section six of this chapter ends with an illustration of the computation of power series solutions for algebraic curves.

Section seven and eight describe prototype modules for a graphical user interface and a computational server. The nineth and last section of this chapter sketches the design of the C interface, the Python interface module, and the wrappers to the C interface to PHCpack.

The last section collects the code snippets, defined for the notebook extension of Jupyter, as they pop up in the menus of the notebook.
3.1 a blackbox solver for isolated solutions

The package phcpy depends on the shared object file phcpy2c2.so for Python2 and phcpy2c3.so for Python3. The module solver exports the blackbox solver of PHCpack, a fast mixed volume calculator, and several functions to predict the number of isolated solutions of a polynomial system. The test_solver() function of the module generates two trinomials (a polynomial with three monomials) with randomly generated complex coefficients.

By default, the input polynomial systems are expected to be square, that is: the number of polynomials in the input list equals the number of variables in the polynomials. The blackbox solver then returns a list of numerical approximations to the isolated solutions of the input polynomial system. Some capabilities of PHCpack to deal with positive dimensional solution sets are exported by the modules sets, cascades, factor, and diagonal. In particular, the solve() function in the factor module computes a numerical irreducible decomposition of the solution set of the polynomial system.

The first of the six subsections describes the basic application of the solve function. The output of solve is a list of strings, with each string representing a solution of the polynomial system given on input. This string representation is explained in the second subsection. The solver depends on the choice of random constants. In the third subsection, the fixing of the seed for the random number generators is demonstrated, for reproducible runs. An important aspect is the construction of a start system, which corresponds to the root counting method. Functions to count the roots in various ways are explained in the fourth subsection. In the fifth subsection, we demonstrate the application of deflation to restore the quadratic convergence of Newton’s method for isolated singularities. Equation and variable scaling improves the numerical conditioning of the solutions, as illustrated in the last subsection.

3.1.1 solving random trinomials and a particular trinomial system

Polynomials and solutions are represented as strings. Below is an illustration of a session with the blackbox solver on a system of two random trinomials, polynomials with three monomials with random complex coefficients.

```
>>> from phcpy.solver import random_trinomials
>>> f = random_trinomials()
>>> for pol in f: print(pol)
```

To solve the system defined by f, we call the blackbox solver:

```
>>> from phcpy.solver import solve
>>> s = solve(f, verbose=False)
>>> len(s)
15
>>> print(s[2])
t : 1.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
x :  7.10290847804173E-01 -4.69841154290980E-01
y : -1.79580717006598E-01 -7.16541556066137E-01
== err : 1.986E-16 = rco : 2.676E-01 = res : 1.232E-16 =
```

The solve command returned a list of 15 strings in s, each string represents a solution that makes the polynomials in f vanish. The module phcpy.solutions (documented in the next section) offers a function to evaluate the solutions in the polynomials given as strings.

By default, the option verbose is set to True and the solver prints the computed root counts. The four computed root counts are

1. The total degree is the product of the degrees of the polynomials in the system.
2. The multi-homogeneous Bézout number is computed on a partition of the set of unknowns.
3. A general **linear-product Bézout number** leads to a start system which is a product of linear polynomials with random coefficients.

4. The **mixed volume** is the mixed volume of the tuple of Newton polytopes of the system. The mixed volume bounds the solutions with all coordinates different from zero. The number of all affine solutions is bounded by the **stable mixed volume**.

For sparse polynomial systems, the mixed volume is a generically sharp root count, i.e.: exact when the coefficients of the polynomial system are sufficiently generic.

Other options of the solver are

1. **tasks**: the number of tasks for multithreaded path tracking. Solving sufficiently large systems on 4 processor cores may result in a speedup of close to a factor 4 if `tasks=4` is given as input argument of `solve`.

   If you do not know how many cores are available on your computer, or if you want to double check, then you can obtain the number of available cores as follows.

   ```python
   >>> from phcpy.phcpy2c3 import py2c_corecount
   >>> py2c_corecount()
   ```

2. **mvfocus**: by default, the blackbox solver computes both Bézout numbers and mixed volumes, because `mvfocus` equals zero. But for sparse polynomial systems, the mixed volume is always significantly smaller than the Bézout numbers. With `mvfocus=1`, the focus on the solver is on mixed volumes and then Bézout numbers will not be computed. For large sparse polynomial systems, using `mvfocus=1` will save time.

3. **precision**: by default the precision is set to `d` for standard hardware double precision. While this precision may suffice, the blackbox solver supports two additional levels of precision: `dd` for double double precision (about 32 decimal places), and `qd` for quad double precision (about 64 decimal places). Given `precision=dd` as extra input parameter to `solve` is likely to yield more accurate results, at an extra cost, which may be compensated by multithreading.

4. **checkin**: by default this flag is set to `True` to check whether the system given on input has as many polynomials as variables. The current version of the blackbox solver accepts only square systems. See the section on positive dimensional solution sets for functions that deal with overdetermined or underdetermined polynomial systems.

Last and certainly not least, the first argument of `solve` is a list of strings. Each string in the list represents a polynomial in several variables. Consider the example below:

```python
>>> from phcpy.solver import solve
>>> p = ['x^2*y^2 + x + 1;', 'x^2*y^2 + y + 1;']
>>> s = solve(p)
total degree : 16
2-homogeneous Bezout number : 8
   with with partition : { x }{ y }
general linear-product Bezout number : 8
   based on the set structure :
   { x }{ x }{ y }{ y }
   { x }{ x }{ y }{ y }
mixed volume : 4
stable mixed volume : 4
```

What is printed to screen by `s = solve(p)` is an example of the four different types of root counts. The structure of the output in `s` is described in the next section.

Because the system `p` is sparse, the mixed volume is much smaller and we can avoid the computation Bézout numbers with the option `mvfocus`, as demonstrated below.

### 3.1. a blackbox solver for isolated solutions
If multitasking is applied in the solver, providing a larger than one value for the option tasks, then the multi-
homogeneous and the general linear-product Bézout numbers are not computed, because of the pipelined polyhedral
homotopies. In pipelined polyhedral homotopies, the computation of the mixed volume is then done by one task, while
the other tasks take the mixed cells and run the polyhedral homotopies to solve a random coefficient start system. This
random coefficient start system will then be used to solve the given system, so there is no need for a start system based
on a Bézout number.

And then there two more options of solve:

5. **dictionary_output**: with polynomials given as strings on input, the solver by default returns the solutions as a
list of strings. With `dictionary_output=True`, the output of `solve` is a list of Python dictionaries. For
example:

```python
>>> from phcpy.solver import solve
>>> p = ['x + y - 1;', 'x - y - 1;']
>>> s = solve(p, dictionary_output=True)
>>> s[0]
{'t': (1+0j), 'm': 1, 'err': 2.22e-16, 'rco': 0.5, 'res': 0.0,
'x': (1+0j), 'y': -0j}
```

The keys in the dictionaries are the names of the variables and other attributes, described in the next section. As
the imaginary unit \( j \) appears, the solver computes with complex numbers.

6. **verbose_level**: by default, the value of the verbose level is zero. Setting `verbose_level` to positive values
will show the names of the modules and procedures in the modules that are called during the blackbox solver. While mainly useful to the software developer, a user may wonder during a long calculation at which stage the
solver appears to be stuck in.

### 3.1.2 representations of solutions of polynomial systems

Solutions of `phcpy.solve` are 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 represent-
ing the polynomials.

The main test in the module solutions is the solution of a small trinomial system and the evaluation of the computed
solutions at the trinomial system.

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. \( \text{err} \): magnitude of the last correction term of Newton’s method (forward error)

4. \( \text{rco} \): estimate for the inverse of the condition number of the Jacobian matrix at the solution
The triplet \((\text{err}, \text{rco}, \text{res})\) measures the numerical quality of the solution. The residual \(\text{res}\) 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 \(\text{rco}\) gives a (sometimes too pessimistic) bound on the number of correct decimal places in the approximate solution. In particular: \(|\text{abs}(\log(\text{rco}, 10))|\) bounds the number of lost decimal places in the approximate solution. For example, if \(\text{rco}\) 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:

```python
>>> from phcpy.solver import solve
>>> p = ['x + y - 1;', 'x - y - 1;']
>>> s = solve(p)
>>> print s[0]
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 \(\text{rco}\) is \(5.0E-1\) which implies that the condition number is bounded by 2, as \(\text{rco}\) 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:

```python
>>> from phcpy.solver import solve
>>> p = ['x + y - 1;', 'x + 0.999*y - 1;']
>>> s = solve(p)
>>> print s[0]
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 \(\text{rco}\) 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:

```python
>>> from phcpy.solver import solve
>>> p = ['x + y - 1;', 'x + 0.999999999*y - 1;']
>>> s = solve(p)
>>> print s[0]
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 =
```

Note 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 \((1,0)\).

3.1. a blackbox solver for isolated solutions
For a solution of the example `noon3` from the module examples, we convert the PHCpack format solution string to a dictionary as follows:

```python
>>> print(s[0])
t : 1.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
x1 : -1.65123467890611E-01 -7.61734168646636E-01
x2 : 8.98653694263692E-01 -3.48820047576431E-01
x3 : 8.98653694263692E-01 -3.48820047576431E-01
== err : 3.034E-16 = rco : 2.761E-01 = res : 5.974E-16 =
```  

```python
>>> from phcpy.solutions import strsol2dict
>>> d = strsol2dict(s[0])
>>> d.keys()
['err', 'res', 'm', 'rco', 't', 'x2', 'x3', 'x1']
>>> d['x1']
(-0.165123467890611-0.761734168646636j)
```

Note 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:

```python
>>> from phcpy.solutions import evaluate
>>> e = evaluate(f,d)
>>> for x in e: print(x)
...
(1.11022302463e-15+4.4408920985e-16j)
(7.77156117238e-16+9.99200722163e-16j)
(7.77156117238e-16+9.99200722163e-16j)
```

A more elaborate verification of the solution is provided by the function `newton_step` of the module `solver` of phcpy.

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.

```python
>>> from phcpy.solutions import make_solution
>>> s0 = make_solution(['x', 'y'], [complex(1, 0), complex(0, 1)])
>>> print(s0)
t : 0.0 0.0
m : 1
the solution for t :
x : 1.000000000000000E+00 0.0
y : 0.000000000000000E+00 1.000000000000000E+00
== err : 0.0 = rco : 1.0 = res : 0.0 ==
>>> s1 = make_solution(['x', 'y'], [float(2), float(3)])
>>> print(s1)
t : 0.0 0.0
m : 1
the solution for t :
x : 2.000000000000000E+00 0.0
y : 3.000000000000000E+00 0.0
== err : 0.0 = rco : 1.0 = res : 0.0 ==
```

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).
```python
>>> from phcpy.solutions import is_real, filter_real
>>> is_real(s0, 1.0e-8)
False
>>> is_real(s1, 1.0e-8)
True
>>> realsols = filter_real([s0, s1], 1.0e-8, 'select')
>>> for sol in realsols: print(sol)
...  
t : 0.0 0.0
m : 1
the solution for t :
x : 2.000000000000000E+00 0.0
y : 3.000000000000000E+00 0.0
== err : 0.0 = rco : 1.0 = res : 0.0 ==
```

The functions `filter_regular` and `filter_zero_coordinates` 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.

```python
>>> from phcpy.solver import solve
>>> p = ['x**2 - 3*y + 1;', 'x*y - 3;']
>>> s = solve(p, verbose=False)
>>> print(s[0])
t : 1.00000000000000E+00 1.14297839516487E+00
m : 1
the solution for t :
x : 1.92017512134718E+00 0.00000000000000E+00
y : 1.56235749888022E+00 9.27337524477545E-124
== err : 2.738E-16 = rco : 2.976E-01 = res : 4.441E-16 =
```

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

```python
>>> from phcpy.solutions import coordinates, make_solution
>>> (names, values) = coordinates(s[0])
>>> names
['x', 'y']
>>> values
[(1.92017512134718+0j), (1.56235749888022+9.27337524477545e-124j)]
>>> s0 = make_solution(names, values)
>>> print(s0)
t : 0.0 0.0
m : 1
the solution for t :
x : 1.920175121347180E+00 0.000000000000000E+00
y : 1.562357498880220E+00 9.273375244775450E-124
== err : 0.0 = rco : 1.0 = res : 0.0 ==
```

Observe that also the diagnostics are set to the defaults.
3.1.3 reproducible runs with fixed seeds

The solver in PHCpack generates different random numbers with each run, which may very well cause the solutions to appear in a different order after a second application of solve on the same system. To prevent this behaviour (to check reproducibility for example), we can fix the seed of the random number generators in PHCpack, as follows:

```python
>>> from phcpy.phcpy2c3 import py2c_set_seed
>>> py2c_set_seed(2013)
0
```

The above session continues as

```python
>>> from phcpy.phcpy2c3 import py2c_get_seed
>>> py2c_get_seed()
2013
```

To reproduce a computation, we can thus request the seed that was used (with `py2c_get_seed`) and then restart the session setting the seed to what was used before (with `py2c_set_seed`).

3.1.4 root counting methods

The performance of the solver is very sensitive to how accurately we can predict the number of solutions. For dense polynomial systems, looking at the highest degrees of the polynomials in the system suffices, whereas for sparse polynomial systems, computing the mixed volume of the Newton polytopes of the polynomials yields much better results. Below is a simple example, illustrating the bounds based on the degrees and the mixed volume:

```python
>>> f = ['x^3*y^2 + x*y^2 + x^2;', 'x^5 + x^2*y^3 + y^2;']
>>> from phcpy.solver import total_degree
>>> total_degree(f)
25
>>> from phcpy.solver import m_homogeneous_bezout_number as mbz
>>> mbz(f)
(19, ' { x }{ y }')
>>> from phcpy.solver import linear_product_root_count as lrc
>>> lrc(f)
a supporting set structure :
{ x }{ x }{ y }{ y }
{ x }{ x }{ x y }{ x y }{ x y }
the root count : 19
19
>>> from phcpy.solver import mixed_volume
>>> mixed_volume(f, stable=True)
(14, 18)
```

The mixed volume is a generically sharp root count for the number of isolated solutions with all coordinates different from zero. The term *generically sharp* means: except for systems with coefficients in a specific collection of algebraic sets, the root count is an exact count. The stable mixed volume counts all affine solutions, that is: also the solutions with zero coordinates. For the example above, we may expect at most 14 isolated solutions with all coordinates different from zero, and, also considering solutions with zero coordinates, at most 18 isolated solutions, counted with multiplicities.

For larger polynomial systems with many different supports, DEMiCs is faster than MixedVol. The code snippet below illustrates the computation of the mixed volume by calling DEMiCs.

```python
>>> f = ['x^3*y^2 + x*y^2 + x^2;', 'x^5 + x^2*y^3 + y^2;']
>>> from phcpy.solver import mixed_volume_by_demics as demics
```
For every root count, total degree, m-homogeneous Bézout number, linear-product root count, and mixed volume, there is a corresponding method to construct a polynomial system with exactly as many regular solutions at the root count, which can then be used as a start system in a homotopy to compute all isolated solutions of the polynomial system for which the root count was computed. Examples of the methods to construct start systems in phcpy are illustrated in the documentation for the module phcpy.trackers.

3.1.5 Newton’s method and deflation

Newton’s method fails when the Jacobian matrix is singular (or close to singular) at a solution. Below is a session on the example of A. Griewank and M. R. Osborne, in their paper Analysis of Newton’s method at irregular singularities, published in SIAM J. Numer. Anal. 20(4): 747-773, 1983. The origin (0,0) is an irregular singularity: Newton’s method fails no matter how close the initial guess is taken. With deflation we can restore the quadratic convergence of Newton’s method:

```python
>>> 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)
t : 0.0 0.0
m : 1
the solution for t :
x : 1.00000000000000E-06 0.0
y : 1.00000000000000E-06 0.0
== err : 0.0 = rco : 1.0 = res : 0.0 ==
>>> from phcpy.solver import newton_step
>>> s2 = newton_step(p,[s])
== err : 1.000E-06 = rco : 5.625E-13 = res : 1.875E-19 =
>>> print(s2[0])
t : 0.00000000000000E+00 0.00000000000000E+00
m : 0
the solution for t :
x : 9.9999996911101E-07 0.00000000000000E+00
y : 9.9999981249806E-13 0.00000000000000E+00
== err : 1.000E-06 = rco : 5.625E-13 = res : 1.875E-19 =
>>> s3 = newton_step(p,s2)
== err : 3.333E-07 = rco : 2.778E-14 = res : 1.111E-13 =
>>> print(s3[0])
t : 0.00000000000000E+00 0.00000000000000E+00
m : 0
the solution for t :
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 =
>>> from phcpy.solver import standard_deflate
>>> sd = standard_deflate(p,[s])
>>> print(sd[0])
t : 0.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
x : -4.55355758042535E-25 2.75154683741089E-26
y : 1.57904709676279E-25 -8.86785799319512E-26
== err : 5.192E-13 = rco : 5.314E-03 = res : 1.388E-16 =
```
The decision to deflate or not depend on the tolerance to decide the numerical rank. Consider the following session:

```python
from phcpy.solutions import make_solution
from phcpy.solver import standard_deflate

sol = make_solution([('x', 'y'), [float(1.0e-6), float(1.0e-6)]),
print(sol)
pols = ['x**2;', 'x*y;', 'y**2;']
sols = standard_deflate(pols, [sol], tolrnk=1.0e-8)
print(sols[0])
sols = standard_deflate(pols, [sol], tolrnk=1.0e-4)
print(sols[0])
```

The default value for `tolrnk` equals `1.0e-6`. If we do not want to deflate that soon, we can lower the tolerance to `1.0e-8` and in that case, there is no deflation when the approximation is still as far as `1.0e-6` from the exact solution. Increasing the value for the tolerance to `1.0e-4` leads to the deflation at the approximation for the solution.

### 3.1.6 the multiplicity of an isolated solution

The multiplicity of an isolated solution can be computed following the ISSAC 2005 paper by Barry Dayton and Zhonggang Zeng on *Computing the multiplicity structure in solving polynomial systems*. Consider again the example of the Griewank-Osborne paper of the previous section:

```python
p = ['(29/16)*x^3 - 2*x*y;', 'x^2 - y;']
from phcpy.solutions import make_solution
s = make_solution([('x', 'y'), [0.0, 0.0])]
from phcpy.solver import standard_multiplicity as multip
print(multip(p,s))
```

The outcome of the commands above is 3, which corresponds to the multiplicity of the isolated solution.

The default value for `order` equals 5, where `order` is the maximal differentiation order. If `order` is too small, then the value on return may be strict lower bound on the multiplicity. Making `order` too large may exhaust the stack size. The default value for the tolerance on the numerical rank is `1.0e-8` and by default `verbose` is set to `False`.

With `dobldobl_multiplicity` and `quaddobl_multiplicity` computations happen respectively in double double and quad double precision.

### 3.1.7 equation and variable scaling

Another source of numerical difficulties are systems that have extreme values as coefficients. With equation and variable scaling we solve an optimization problem to find coordinate transformations that lead to better values for the coefficients. The common sense approach to scaling is described in Chapter 5 of the book of Alexander Morgan on *Solving Polynomial Systems Using Continuation for Engineering and Scientific Problems*, volume 57 in the SIAM Classics in Applied Mathematics, 2009. We consider a simple example.

```python
>>> 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, verbose=False)
>>> print(psols[0])
t : 1.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
x : -3.23606797749979E+03 8.71618409420601E-19
y : 2.30490982555757E-19 1.27201964951407E+03
```
Observe the rather large values of the coordinates in the first solution and the estimate for the inverse condition number. We scale the system as follows:

```python
>>> from phcpy.solver import standard_scale_system as scalesys
>>> from phcpy.solver import standard_scale_solutions as scalesols

>>> (q, c) = scalesys(p)
>>> q[0]
'x^2 + 9.99999999999998E-01*y^2 - 1.00000000000000E+00;'
>>> q[1]
'y^2 - 1.00000000000000E+00*x;'
```

The coefficients in the scaled system look indeed a lot nicer. In the parameter `c` returned along with the scaled system are the scaling coefficients, which we need to bring the solutions of the scaled system into the original coordinates.

```python
>>> qsols = solve(q, verbose=False)
>>> ssols = scalesols(len(q), qsols, c)

>>> for sol in ssols: print(sol)
...  
t : 1.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
x : -3.23606797749978E+03 -1.98276706040285E-115
y : 0.00000000000000E+00 -1.27201964951407E+03
== err : 1.746E-16 = rco : 2.268E-01 = res : 2.220E-16 =
t : 1.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
x : -3.23606797749978E+03 -1.98276706040285E-115
y : 0.00000000000000E+00 1.27201964951407E+03
== err : 1.746E-16 = rco : 2.268E-01 = res : 2.220E-16 =
t : 1.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
x : 1.23606797749979E+03 0.00000000000000E+00
y : 7.86151377757423E+02 0.00000000000000E+00
== err : 4.061E-17 = rco : 4.601E-01 = res : 5.551E-17 =
t : 1.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
x : 1.23606797749979E+03 0.00000000000000E+00
y : -7.86151377757423E+02 7.38638289422858E-124
== err : 4.061E-17 = rco : 4.601E-01 = res : 5.551E-17 =
```

The estimates of the condition numbers in `ssols` are for the scaled problem. With scaling, the condition numbers were reduced from $10^4$ to 10. For more extreme values of the coefficients, we may have to perform the scaling in higher precision, such as available in the functions `dobldobl_scale_system` and `quaddobl_scale_system`, respectively with double double and quad double arithmetic.
3.1.8 reduction of polynomial systems

Applying row reduction on the coefficient matrix of a polynomial system may lead to a system with fewer monomials and a lower root count. Consider for example the following session:

```python
>>> p = ['x**2*y**2 + x + 1;', 'x**2*y**2 + y + 1;']
>>> from phcpy.solver import linear_reduce
>>> r = linear_reduce(p)
>>> for pol in r: print(pol)
```

The printed polynomials are $x^2y^2 + y + 1$; and $+ x - y$; showing that, while the system is invariant under swapping of $x$ and $y$, all solutions are fixed points as both coordinates for all four solutions will be the same.

The precision of the row reduction is increased to double double by providing the argument `precision='dd'` and to quad double via the argument `precision='qd'`.

Nonlinear reduction computes S-polynomials to eliminate the leading term and then, if a criterion with R-polynomials is satisfied, replaces one of the polynomials in the system by the S-polynomial. Consider the session below:

```python
>>> from phcpy.solver import standard_nonlinear_reduction as reduce
>>> pols = ['x^3 - x;', 'x^2*y + 1;']
>>> redu = reduce(pols)
```

What is printed are the polynomials $+ y + 1$; and $x^2 - 1$; which allows to read off the solutions.

3.2 path trackers and sweep homotopies

Homotopy continuation methods are applied to solve a polynomial system. The module `phcpy.trackers` exports the path trackers of PHCpack. The functions in this module track paths defined by artificial-parameter homotopies, of the form

$$h(x, t) = \gamma(1 - t)g(x) + tf(x) = 0,$$

where $\gamma$ is a randomly generated complex constant. The artificial parameter $t$ goes from zero to one, from the known solutions of the start system $g(x) = 0$ to the solutions of the target system $f(x) = 0$.

The module `phcpy.sweepers` exports the sweep homotopies. A sweep homotopy is a natural parameter homotopy. Its application is to track solution paths from one set of values for the parameters to another set of values for the parameters.

The tracking of solution paths defined by an artificial-parameter homotopy apply an increment-and-fix method: the continuation parameter $t$ is incremented by the predictor and remains fixed in the corrector. The tracking of solution paths defined by a sweep homotopy apply arc length parameter continuation.
3.2.1 a simple example

The example session below illustrates the computation of the intersection of an ellipse with a parabola. A homotopy method based on the total degree replaces the two given quadratic equations for the ellipse and the parabola by a configuration of lines that has exactly as many solutions as the expected number of intersection points. The homotopy connects the given system with the equations of the simpler configuration, which define the start system. Continuation methods track the paths starting at the solutions of the start system to the solutions of the target system.

```python
>>> from phcpy.solver import total_degree
>>> from phcpy.solver import total_degree_start_system
>>> from phcpy.trackers import track

>>> p = ['x^2 + 4*y^2 - 4;', '2*y^2 - x;']
>>> d = total_degree(p)

4

>>> (q, qsols) = total_degree_start_system(p)

4

>>> q
['x^2 - 1;', 'y^2 - 1;']

>>> s = track(p, q, qsols)

4

>>> for sol in s: print(sol)
...
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 : 1.309E-16 = rco : 1.998E-01 = res : 4.441E-16 =
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 : 1.309E-16 = rco : 1.998E-01 = res : 4.441E-16 =
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 : 1.505E-36 = rco : 1.079E-01 = res : 0.000E+00 =
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 : 1.505E-36 = rco : 1.079E-01 = res : 0.000E+00 =
```

As expected when we intersect two quadratic equations, we find four intersection points. The coordinates of the solutions are complex numbers, listed as two consecutive floating-point numbers in scientific notation. The two consecutive numbers approximate the real and imaginary part of the complex number. In the four solutions above, observe that two solutions are real and two solutions are complex conjugate.

Note that the start system `q` in `['x^2 - 1;', 'y^2 - 1;']` has four real solutions, while the system `p` we solve had two complex conjugate solutions. If we connect `p` to `q` with a real homotopy, then at some point along the path, two real solutions have to turn into a pair of complex conjugate solutions. Multiplying the start system with a
random complex constant, we avoid the singularities along the solution paths. The side effect of this multiplication is that different constants will result in different orders of the solutions at the end. For example:

```python
>>> from phcpy.solver import total_degree_start_system
>>> from phcpy.trackers import track

>>> p = ['x^2 + 4*y^2 - 4;', '2*y^2 - x;']
>>> (q, qsols) = total_degree_start_system(p)
>>> s1 = track(p, q, [qsols[2]])
>>> print(s1[0])
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 : 1.383E-16 = rco : 1.998E-01 = res : 2.220E-16 =

```n

```python
>>> s2 = track(p, q, [qsols[2]])
>>> print(s2[0])
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 : 4.815E-35 = rco : 1.079E-01 = res : 0.000E+00 =
```n

3.2.2 fixing the gamma constant

To avoid this side effect, `track` accepts a complex value as its last argument for the so-called gamma constant. As a continuation of the session from above:

```python
>>> s3 = track(p, q, [qsols[2]], gamma=complex(0.824372806319,0.56604723848934))
>>> print(s3[0])
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 : 4.815E-35 = rco : 1.079E-01 = res : 0.000E+00 =
```

If we track all solution paths one after the other, each time calling `track` with the same value for gamma, then all solutions will be found.

3.2.3 give the next solution on a path

The `track` function follows a solution path till the end. Often it could be useful to view all intermediate solutions computed along a path. The functions `next_standard_solution()`, `next_dobldobl_solution()`, `next_quaddobl_solution()`, and `next_multprec_solution()`, implement generators for the path trackers in standard double, double double, quad double precision, and arbitrary multiprecision respectively. With these `next_` functions, the user not only gets all solutions along a path, but also receives control of the order of execution. Before the application of `next_`, one must initialize the homotopy with target and start system and give an initial start solution. The session below illustrates the use of this generator:

```python
>>> from phcpy.solver 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_standard_tracker
from phcpy.trackers import initialize_standard_solution
from phcpy.trackers import next_standard_solution

initialize_standard_tracker(p, q)
initialize_standard_solution(len(p), s[0])
s1 = next_standard_solution()
print(s1)
t : 1.00000000000000E-01 0.00000000000000E+00
m : 1
the solution for t :
x : 9.9633843834030E-01 4.70831004481527E-03
y : 9.96408320626402E-01 4.95310952563875E-03
== err : 2.375E-05 = rco : 1.000E+00 = res : 3.619E-10 =

print(next_standard_solution())
t : 2.00000000000000E-01 0.00000000000000E+00
m : 1
the solution for t :
x : 9.80919860804043E-01 1.78496473654540E-02
y : 9.81218221286503E-01 2.32056259678926E-02
== err : 1.671E-08 = rco : 1.000E+00 = res : 1.424E-16 =

print(next_standard_solution())
t : 3.00000000000000E-01 0.00000000000000E+00
m : 1
the solution for t :
x : 9.51909891692765E-01 2.71534790078036E-02
y : 9.42895891640611E-01 5.51080014180090E-02
== err : 4.812E-09 = rco : 1.000E+00 = res : 1.665E-16 =

In the session above, we see the solutions s1 for t = 0.1, and two other solutions for consecutive values 0.2 and 0.3 for t. If we continue the session from above with the second start solution in s[1], we can select the first 11 points along the path and view all values for x of the solutions:

initialize_standard_solution(len(p), s[1])
points = [next_standard_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]
for x in xvals: print(x)
...
(0.996338438384+0.00470831004482j)
(0.980919860804+0.0178496473655j)
(0.951909891693+0.0271534790078j)
(0.924234166108+0.0231054530961j)
(0.908102639672+0.014159812703j)
(0.90039366434+0.00726313574566j)
(0.896843555845+0.00320608226584j)
(0.895239133202+0.00112430968375j)
(0.89458634218+0.000224845127444j)
(0.894427191+2.20881053462e-28j)
(0.894427191+0j)

We see that the last two values differ little from each other because we arrived at the end of the path. To test whether at the end of a path, it suffices to check whether the value for t equals one.

The real parts of the four paths are shown in Fig. 3.1 Three of the paths converge to the triple solution (1,2).

The code used to make the plot (using matplotlib) is below:

3.2. path trackers and sweep homotopies
Fig. 3.1: The real parts of four solution paths.
```python
p = ['x^2 + y - 3;', 'x + 0.125*y^2 - 1.5;']
print('constructing a total degree start system ...')
from phcpy.solver import total_degree_start_system as tds
q, qsols = tds(p)
print('number of start solutions :', len(qsols))
from phcpy.trackers import initialize_standard_tracker
from phcpy.trackers import initialize_standard_solution
from phcpy.trackers import next_standard_solution
initialize_standard_tracker(p, q, False)
from phcpy.solutions import strsol2dict
import matplotlib.pyplot as plt
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]
deinit = strsol2dict(startsol)
xpoints = [deinit['x']]
ypoints = [deinit['y']]""
for k in range(300):
    ns = next_standard_solution()
    deinit = strsol2dict(ns)
    xpoints.append(deinit['x'])
    ypoints.append(deinit['y'])
tval = deinit['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_xlim(min(yre)-0.3, max(yre)+0.3)
dots, = axs.plot(xre,yre,'r-')
fig.canvas.draw()
fig.canvas.draw()
an = raw_input('hit return to exit')
```

With `False` in `initialize_standard_tracker(p, q, False)` the option to generate a fixed gamma constant is turned off, so rerunning the same code will generate other random constants and produce different plots.

To set the value of the gamma constant to a specific value, e.g.: 

```
(-0.853618933016554-0.52089799116111j)
```

Do the following.

```python
gamma = (-0.853618933016554-0.52089799116111j)
initialize_standard_tracker(target, start, False, gamma.real, gamma.imag)
```

3.2. path trackers and sweep homotopies
3.2.4 solving with polyhedral homotopies

Below is an interactive session to illustrate the solving with polyhedral homotopies.

```python
>>> p = ['x^3*y^2 - 3*x^3 + 7;', 'x*y^3 + 6*y^3 - 9;']
>>> from phcpy.solver import mixed_volume
>>> mixed_volume(p)
11
>>> from phcpy.solver import random_coefficient_system
>>> (q, qsols) = random_coefficient_system(silent=True)
>>> len(qsols)
11
>>> from phcpy.trackers import track
>>> psols = track(p, q, qsols)
>>> len(psols)
11
>>> print(psols[4])
t : 1.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
x : -7.33932797408386E-01 -9.84310202527377E-01
y : -6.56632351304388E-01 9.90969278772793E-01
-- err : 1.938E-16 = rco : 5.402E-01 = res : 2.102E-15 =
```

3.2.5 Newton's method at higher precision

We can apply one Newton step with higher precision to improve the accuracy of the solutions. Doubling the precision:

```python
>>> psols_dd = newton_step(p, psols, precision='dd')
== err : 5.190E-15 = rco : 3.266E-03 = res : 3.342E-27 =
== err : 5.051E-15 = rco : 1.978E-02 = res : 2.727E-28 =
== err : 5.015E-16 = rco : 5.402E-01 = res : 3.525E-30 =
We see that the residual (the parameter res) drops for every solution.
```

Below is an illustration of the use of linear-product start systems:

```python
>>> p = ['x*y^3 + y - 2;', 'x^3*y + x - 8;']
>>> from phcpy.solver import linear_product_root_count
>>> r = linear_product_root_count(p)
a supporting set structure :
   { x }{ y }{ y }{ y }
   { x }{ x }{ x }{ y }
the root count : 10
>>> from phcpy.solver import random_linear_product_system
>>> (q, qsols) = random_linear_product_system(p)
>>> len(qsols)
10
(continues on next page)
```
Looking at the values for \textit{err} and \textit{res} we see huge values for two solutions which are spurious.

### 3.2.6 multitasked path tracking

Last but certainly not least, consider the application of multitasking to path tracking. On the benchmark problem of cyclic 7-roots:

```bash
$ time python trackcyclic7.py
number of start solutions : 924
starting the path tracking with 1 task(s) ...
tracked 924 solution paths

real  0m7.147s
user  0m7.126s
sys   0m0.016s

$ time python trackcyclic7.py 2
number of start solutions : 924
starting the path tracking with 2 task(s) ...
tracked 924 solution paths

real  0m3.927s
user  0m7.640s
sys   0m0.017s
```

Observe that the wall clock time (the time following the \textit{real}), is cut almost in half when 2 tasks are used. The script is below:

```python
from sys import argv
if(len(argv) == 1):
    nbtasks = 1
else:
    nbtasks = eval(argv[1])
from phcpy.phcpy2c import py2c_read_standard_target_system_from_file
from phcpy.phcpy2c import py2c_read_standard_start_system_from_file
from phcpy.phcpy2c import py2c_copy_target_system_to_container
from phcpy.phcpy2c import py2c_copy_start_system_to_container
from phcpy.phcpy2c import py2c_copy_start_solutions_to_container
from phcpy.phcpy2c import py2c_solcon_number_of_solutions
```

(continues on next page)
3.2.7 GPU accelerated path tracking

The script below illustrates the call to the GPU accelerated path trackers. As input, the location of a random coefficient system (as obtained via the polyhedral homotopies) is needed. With this random coefficient system, we have an artificial-parameter homotopy that defines 35,940 solution paths to solve the cyclic 10-roots problem.

```python
GPU = 1  # use the GPU
DIR = '/home/jan/Problems/GPUdata/MultiPath'  # location of systems
from phcpy.phcpy2c\nimport py2c_read_standard_target_system_from_file as read_target
from phcpy.phcpy2c\nimport py2c_read_standard_start_system_from_file as read_start
cyc10tarfile = DIR + '/cyclic10.target'
cyc10stafile = DIR + '/cyclic10.start'
fail = read_target(len(cyc10tarfile), cyc10tarfile)
from phcpy.interface import load_standard_system as loadsys
cyc10 = loadsys()
print('the cyclic 10-roots problem :')
for pol in cyc10:
    print(pol)
fail = read_start(len(cyc10stafile), cyc10stafile)
cyc10q = loadsys()
print('a start system for the cyclic 10-roots problem :')
for pol in cyc10q:
    print(pol)
cyc10qsols = loadsys()
print('number of start solutions :', len(cyc10qsols))
if (GPU == 0):
    from phcpy.trackers import ade_double_track
cyc10sols = ade_double_track(cyc10, cyc10q, cyc10qsols, verbose=0)
else:
    from phcpy.trackers import gpu_double_track
cyc10sols = gpu_double_track(cyc10, cyc10q, cyc10qsols, verbose=0)
print('number of solutions :', len(cyc10sols))
```

(continues on next page)
for sol in cyc10sols:
    print(sol)

### 3.2.8 sweep homotopies

A *sweep homotopy* is a family of polynomial systems with at least one natural parameter and one artificial parameter. As the artificial parameter moves from zero to one, the natural parameter changes from a given start value to another given target value. By arc length continuation, the solution paths are tracked from the given start values for the parameters to the target values.

Consider a simple example: sweeping the circle. We consider the unit circle $x^2 + y^2 - 1 = 0$, intersected by a horizontal line, at the start equal to $y = 0$. In a Python session, we could define the sweep homotopy that takes the line from $y = 0$ to $y = 2$.

```python
>>> circle = ['x^2 + y^2 - 1;', 'y*(1-s) + (y-2)*s;']
```

For $s = 0$ there are two solutions, with values for $x$ and $y$ in the tuples $(1, 0)$ and $(-1, 0)$.

Geometrically, as the horizontal line moves up, the two solutions (the intersection points on the circle and the line), move closer to each other to join at a *quadratic turning point*, shown in Fig. 3.2. At the left picture of Fig. 3.2 we see the line transversally intersecting the circle at a perfect right angle. At the right picture of Fig. 3.2, the two distinct solutions have merged into one point where the line is tangent to the circle.

![Fig. 3.2: Two complex conjugated solutions meet at a quadratic turning point.](image)

The tracking of solution paths in a real sweep homotopy will stop at the first singular point it encounters. The continuation of the code with the definition of `circle` to launch this path tracking is listed below:

```python
>>> from phcpy.solutions import make_solution as makesol
>>> first = makesol(['x', 'y', 's'], [1, 0, 0])
>>> second = makesol(['x', 'y', 's'], [-1, 0, 0])
>>> startsols = [first, second]
>>> from phcpy.sweepers import standard_real_sweep as sweep
>>> newsols = sweep(circle, startsols)
>>> print(newsols[0])
```

and then we see as output of the `print` statement:

```
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
```
The sweep stopped where $s$ is equal to 0.5, with corresponding values for $x$ and $y$ in the tuple $(0, 1)$.

### 3.2.9 real versus complex sweeps

In a complex sweep, an addition random gamma constant is generated in the convex-linear combination between the sets of start and target values for the parameters. If the solutions for the start values of the parameters are regular, then the application of the gamma trick applies for problems where the parameter space is convex. This means that, if the problem formulation makes sense for convex combinations of the parameters, then the solution paths will remain regular, except for finitely many bad choices of the random gamma constant, and except perhaps at the very end of the paths, when the target values for the parameters lead to polynomial systems with singular solutions.

Conducting a complex sweep on the circle can be done as follows:

```python
>>> circle = ['x^2 + y^2 - 1;']
>>> from phcpy.solutions import make_solution as makesol
>>> first = makesol(['x', 'y'], [1, 0])
>>> second = makesol(['x', 'y'], [-1, 0])
>>> startsols = [first, second]
>>> par = ['y']
>>> start = [0, 0]
>>> target = [2, 0]
>>> from phcpy.sweepers import standard_complex_sweep as sweep
>>> newsols = sweep(circle, startsols, 2, par, start, target)
```

The setup of the homotopy defines $y$ as the parameter (in the list ['y'] assigned to `par`). The parameter $y$ will move from the complex zero $0 + 0i$ (given by the list [0, 0] assigned to `start`) to $2 + 0i$ (given by the list [2, 0] assigned to `target`). The corresponding start solutions for $y = 0$ are stored in the tuples $(1, 0)$ and $(-1, 0)$. Then, at the end of the sweep, we will find two complex conjugated solutions.

```python
>>> print(newsols[0])
t : 1.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
x : -6.27554627321419E-26 1.73205080756888E+00
y : 2.00000000000000E+00 0.00000000000000E+00
== err : 6.642E-13 = rc0 : 1.000E+00 = res : 4.441E-16 =
```

What is now the difference between real versus complex? The real sweep stopped at the singular solution $(0, 1)$ while the complex sweep hopped over this singularity because of complex random gamma constant in the convex combination between the start and target values of the parameters.
3.2.10 tuning parameters, settings, and tolerances

The default values of the numerical parameters were set based on computational experiences on a large, representative collection of polynomial systems. The module tuning provides functions to adjust the parameters, settings, and tolerances. The function tune_track_parameters gives access to the tuning as in phc -p, via an interactive menu. The other functions in the module allow to get the values and to set the values of each parameter, setting, or tolerance.

3.2.11 a polyhedral end game

In case the mixed volume is not a sharp root count, there are paths diverging to points with coordinates equal to zero, or diverging to infinity. The directions of those diverging paths coincide with the leading exponents of the Puiseux series expansions of the points with coordinates equal to zero and/or at infinity. In particular, positive leading exponents occur with coordinates going to zero, while for a coordinate at infinity, the corresponding leading exponent will be negative.

To activate the polyhedral end game, the extrapolation order needs to be nonzero. We can set this order as follows:

```python
>>> from phcpy.tuning import order_endgame_extrapolator_set as set
>>> set(4)
```

The 0 on return is the failure code, which should equal zero if all went well. To double check, we can get the value of the order of the extrapolator in the end game:

```python
>>> from phcpy.tuning import order_endgame_extrapolator_get as get
>>> get()
```

Let us run a polyhedral end game on a very simple example.

```python
>>> f = ['x + y^3 - 1;', 'x + y^3 + 1;']
>>> from phcpy.solver import mixed_volume as mv
>>> from phcpy.solver import random_coefficient_system as rcs
>>> mv(f)
4
>>> (g, gsols) = rcs(f)
>>> len(gsols)
4
```

Although the mixed volume equals four (and we have four start solutions in gsols of the start system g), we can see that f has no solutions, and all four paths will diverge to infinity.

```python
>>> from phcpy.trackers import standard_double_track as track
>>> sols = track(f, g, gsols)
>>> from phcpy.tropisms import standard_retrieve as retrieve
>>> (w, d, e) = retrieve(len(sols), len(f))
>>> w
[3, 3, 3, 3]
```

We see that the winding numbers of the four paths are all equal to 3 and the numerically computed tropisms are approximations of (-1, -1/3), or (-3, -1) when presented in normal form.
### 3.3 positive dimensional solution sets

The modules *sets*, *cascades*, *factor*, and *diagonal* provide some functionality to work with positive dimensional solution sets. In particular, the *solve* function of the *factor* module computes a numerical irreducible decomposition of the solution set of a polynomial system. Also polynomials that have variables raised to negative powers, so-called *Laurent polynomials* are supported.

For isolated solutions, the main outcome of the numerical solver is a list of points, given as tuples of values for the coordinates. For positive dimensional solutions, with numerical homotopy continuation methods we can compute a *numerical irreducible decomposition* of the solution set. Such a decomposition has two layers:

1. For every dimension of the solution set, we have as many generic points as the degree of the solution set of that dimension.
2. For every dimension of the solution set, those generic points that belong to the same irreducible factor are stored in the same list.

A *generic point* on a $d$-dimensional solution set is computed as a solution of the given system of polynomial equations, augmented with $d$ linear equations with randomly generated complex coefficients.

Generic points occur as solutions in the data structure that is called a witness set. With embeddings and cascades, we define homotopies in a top down calculation of a numerical irreducible decomposition. Diagonal homotopies define a bottom up construction of a numerical irreducible decomposition. The application of monodromy loops leads to a factorization of a pure dimensional solution set into irreducible components. Even without having explicit equations for the irreducible factors, with a homotopy membership test we can determine whether any given point belongs to any given factor in the decomposition.

#### 3.3.1 witness sets

A *witness set* is a data structure to represent a positive dimensional solution set, which is stored as a tuple of two items:

1. An *embedding* of the polynomial equations that define the solution set, augmented with as many generic linear equations as the dimension of the solution set. To every linear equation corresponds one *slack variable*.
2. Witness points are solutions in the intersection of the original polynomial equations and the generic linear equations. For generic coefficients of the added linear equations, we obtain *generic points* on the solution set. The number of witness points equals the degree of the solution set.

In the example below we consider the twisted cubic:

```python
>>> twisted = ['x^2 - y;', 'x^3 - z;']
>>> from phcpy.sets import embed
>>> e = embed(3,1,twisted)
>>> e[0]
'x^2 - y + (-8.23538851649530E-01-5.67259869745581E-01*i)*zz1;'
>>> e[1]
'x^3 - z + (9.35464826338593E-01-3.53419805165623E-01*i)*zz1;'
```

The last equation of the embedded system is a linear equation with randomly generated complex coefficient. The *zz1* denotes the slack variable. Continuing the session:

```python
>>> terms = e[-1].split(')*')
>>> for t in terms: print(t)
... + (-8.85038627286137E-01 + 4.65517591731472E-01*i
x + (-2.12324313395875E-02 + 9.99774566519578E-01*i
y + (-9.52478263619098E-01-3.04606561539880E-01*i
```

(continues on next page)
The variable \(zz1\) is an artificial slack variable. Adding the slack variable via an embedding is a general technique to make overdetermined polynomial systems \textit{square}, that is: having as many equations as unknowns. Only solutions with zero slack variables matter.

There are four homotopies which involve witness sets.

1. Given a witness set and a point, a \textit{homotopy membership test} decides whether the point lies on the solution set represented by the witness set.
2. Given all solutions with nonzero values for the slack variables of an embedded system, a \textit{cascade homotopy} takes those solutions as the start points of solution paths leading to generic points on lower dimensional solution sets.
3. Given a witness set, a \textit{monodromy homotopy} separates the generic points in the witness set according to the irreducible factors of the solution set.
4. Given two witness sets, a \textit{diagonal homotopy} computes witness set representations for all components of the intersection of the two given witness sets.
3.3.2 homotopy membership test

Given a witness set and a point, with a homotopy we can decide whether the point belongs to the algebraic set represented by the given witness set. We illustrate this membership test on the cyclic 4-roots problem. First we compute a witness set.

```python
>>> from phcpy.families import cyclic
>>> c4 = cyclic(4)
>>> from phcpy.sets import embed
>>> c4e1 = 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')
>>> for sol in genpts:
...    print(sol)
Because there are four solutions that satisfy the original cyclic 4-roots problem and a hyperplane with randomly generated coefficients, there is a one dimensional solution set of cyclic 4-roots.

The function `membertest` takes as input the witness set, represented by the polynomials in `c4e1` and the generic points in `genpts`, and a point. The point is given as a list of doubles, with the real and imaginary parts of all coordinates. The point `(1, -1, 1, -1)` is thus given as the list `[1, 0, -1, 0, 1, 0, -1, 0]`. The four extra zeroes are the zero imaginary parts of the four coordinates.

```python
>>> point = [1, 0, -1, 0, 1, 0, -1, 0]
>>> from phcpy.sets import membertest
>>> membertest(c4e1, genpts, 1, point)
residual is 4.00000000000000E+00
point does not lie on the component, as residual > 1.000E-06
False
```

The function `membertest` returns `False` as the residual of the evaluation of the point at the equations does not satisfy the default tolerance.

Testing the point `(-1, -1, 0, 0)` proceeds as follows. The ... below stands for omitted output.

```python
>>> point = [-1, 0, -1, 0, 1, 0, 1, 0]
>>> membertest(c4e1, genpts, 1, point)
residual is 0.00000000000000E+00
point satisfies the equations, as residual <= 1.000E-06
```

The point passes the residual test. The test continues with the computation of new generic points for a hyperplane that passes through the test point. If the test point is among the new generic points, then the test point belongs to the

The point passes the residual test. The test continues with the computation of new generic points for a hyperplane that passes through the test point. If the test point is among the new generic points, then the test point belongs to the
positive dimensional solution set represented by the witness set. For this example we see that the point \((-1, -1, 1, 1)\) is a singular point on the curve, as can be seen from the estimate for the inverse condition number, \(rco : 2.239E-09\). The default tolerance of \(1.0e-6\) is high enough in this case for the point to satisfy the membership test.

If the tolerance \(1.0e-6\) is deemed too sloppy, then we can allow for a stronger tolerance and execute the homotopy membership test in double double precision. More zeroes must be inserted in the test point for the second part (the least significant double) in the double double representation for the real and imaginary parts of the coordinates:

```python
>>> ddpoint = [-1, 0, 0, 0, -1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]
```

Instead of \(1.0e-6\), the new tolerance is \(1.0e-12\):

```python
>>> membertest(c4e1, genpts, 1, ddpoint, memtol=1.e-12, precision='dd')
residual is 0.00000000000000000000000000000000E+00
point satisfies the equations, as residual <= 1.000E-06
...
== 3 = #step : 43 #fail : 9 #iter : 119 = singular solution
t : 1.00000000000000000000000000000000E+00 0.00000000000000000000000000000000E+00
m : 1 Length of path : 2.58366452243257E+01
the solution for t :
x0 : -1.00000000000000000000460097514793E+00 -5.0354651555782583675842894198701E-21
x1 : -9.99999999999999999999596789412885036927768E-21 5.03544121195596789412885036927768E-21
x2 : 9.999999999999999999995602319048E-01 4.8488723658967816181430534515216E-24
x3 : 1.00000000000000000000000000000000E+00 4.872815984264759732685155434612E-24
zz1 : 3.21588798303478472454372240227796E-34 -4.6127068218274744272903697352050E-34
== err : 1.525E-13 = rco : 1.343E-14 = res : 8.687E-26 ==
... match with generic point 3, as difference is 4.183E-17 <= 1.000E-12
Point lies on a solution component.
```

In double double precision, the condition number estimate for the inverse condition number drops to \(1.343E-14\) (see the \(rco\) field).

To perform the membership test in quad double precision, invoke `membertest` with `precision='qd'`.

For solution sets of large degree, the homotopy membership test will run faster in its multitasked version. To run the membership test with 8 tasks, add `tasks=8` as last argument of the call to the function.

### 3.3.3 cascade of homotopies

With a cascade of homotopies, we separate generic points on one equidimensional component from another equidimensional component of the solution set. A cascade starts at the top dimension. We consider an illustrative example:

```python
>>> pols = ['(x^2 + y^2 + z^2 - 1)*(y - x^2)*(x - 0.5);',
' (x^2 + y^2 + z^2 - 1)*(z - x^3)*(y - 0.5);',
' (x^2 + y^2 + z^2 - 1)*(z - x*y)*(z - 0.5);']
```

The polynomials in `pols` are defined in factored form so for this illustrative example we may read of the equidimensional components of the solution set, which contain the two dimensional sphere, the one dimensional twisted cubic, and the isolated point \((0.5, 0.5, 0.5)\).

To initialize the cascade, we must have solved an embedded polynomial system. With `embed(3, 2, pols)` we make an embedding of the 3-dimensional system in `pols` adding two linear equations with random complex coefficients. Two slack variables `zz1` and `zz2` are added to make this overdetermined system square.
The list `topsols` contains two types of solutions: those with nonzero values for the slack variables, and those with zero slack variables, which thus satisfy the original equations in `pols` and the two added linear equations with random complex coefficients. The solutions with zero values for the slack variables define generic points on the two dimensional solution set. We filter the solutions, as follows:

```python
>>> from phcpy.solutions import filter_zero_coordinates as filter
>>> topsols0 = filter(topsols, 'zz2', 1.0e-8, 'select')
>>> topsols1 = filter(topsols, 'zz2', 1.0e-8, 'remove')
>>> for sol in topsols0:
...     print(sol)
```

The solutions with nonzero values for the slack variables are called *nonsolutions*. These solutions are regular and serve as start solutions in a cascade to compute generic points on the lower dimensional components of the solution set.

```python
>>> from phcpy.cascades import cascade_step
>>> lvl1sols = cascade_step(2, topemb, topsols1)
```

After the filtering, we must drop variables, coordinates, and hyperplane for the next level in the cascade.

```python
>>> from phcpy.sets import drop_variable_from_polynomials as drop1poly
>>> from phcpy.sets import drop_coordinate_from_solutions as drop1sols
>>> lvl1emb = drop1poly(topemb, 'zz2')
>>> lvl1emb = lvl1emb[:-1]  # dropping the last polynomial
>>> lvl1solsdrop = drop1sols(lvl1sols, len(topemb), 'zz2')
>>> lvl1sols0 = filter(lvl1solsdrop, 'zz1', 1.0e-8, 'select')
>>> lvl1sols1 = filter(lvl1solsdrop, 'zz1', 1.0e-8, 'remove')

Among the solutions at the end of the paths defined by the cascade homotopy are solutions that belong to the two dimensional sphere. These solutions are singular and we filter then away based on threshold for the estimate of the inverse condition number.

```python
>>> from phcpy.solutions import filter_regular as regfilt
>>> reglvl1sols0 = regfilt(lvl1sols0, 1.0e-8, 'select')
>>> for sol in reglvl1sols0:
...     print(sol)
```

To find the isolated solutions, another cascade homotopy is applied, tracking the paths starting at the nonsolutions at the end of the previous cascade.

```python
>>> lvl2sols = cascade_step(1, lvl1emb, lvl1sols1)
>>> lvl2solsdrop = drop1sols(lvl2sols, len(lvl1emb), 'zz1')
>>> for sol in reglvl2solsdrop:
...     print(sol)
```

To perform the filtering of the solutions properly, we apply a membership test, defined in the *sets* module.

The function `run_cascade()` takes on input the number of variables in the polynomials and the top dimension of the solution set. Starting at the top dimension, a witness set representation for each pure dimensional component of the solution set is computed.
## 3.3.4 factoring into irreducibles

A witness set consists of two parts. The first part of a witness set is a polynomial system with as many added linear equations with random coefficients as the dimension. The number of slack variables (variables that start with the name `zz`) equals the dimension of the witness set. The second part of a witness set is a list of solutions of the first part. Because the added linear equations have random coefficients, the solutions are generic points on the positive dimensional algebraic set.

Given a witness set, applying monodromy loops those points in a witness set that lie on the same irreducible factor are joined. The application of monodromy is a probabilistic method with unknown probability of failure because it relies on the unknown distribution of the singular solutions.

Below is a simple example, given already in factored form:

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

To construct a witness set we import `witness_set_of_hypersurface` from `phcpy.sets`:

```python
>>> from phcpy.sets import witness_set_of_hypersurface as wh
>>> (w, s) = wh(2, p)
>>> len(s)
```

Because the degree of `p` is three, we see 3 as the outcome of `len(s)`.

```
>>> from phcpy.factor import factor
>>> f = factor(1, w, s)
>>> f
```

The result in `f` is a a list of tuples:

```
[[(1, 2), 8.537360146292391e-15], ([3], 2.1316282072803006e-14)]
```

The factorization joined the first two solutions of `s` as they represent the quadratic factor. A generic point for the linear factor is in the second tuple. The second floating point number in each tuple is the residual obtained via the linear trace test, used as stop criterion in the running of monodromy loops.

For polynomials of higher degrees, double double or even quad double could be required to obtain accurate results. The following two commands illustrate how to apply monodromy respectively in double double and quad double precision:

```
>>> f = factor(1, w, s, precision='dd')
>>> f = factor(1, w, s, precision='qd')
```

The witness set `(w, s)` should also have been computed in double double and quad double precision.

The function `decompose()` takes the output of the `run_cascade()` function of the `cascades` module and factors every witness set for the pure dimensional components into irreducible factors. The functions `run_cascade()` and `decompose()` lead to a numerical irreducible decomposition of the solution set.
### 3.3.5 numerical irreducible decomposition

Consider the polynomials defined by the list `pols` as follows:

```python
>>> pols = ['(x1-1)*(x1-2)*(x1-3)*(x1-4);',
          '(x1-1)*(x2-1)*(x2-2)*(x2-3);',
          '(x1-1)*(x1-2)*(x3-1)*(x3-2);',
          '(x1-1)*(x2-1)*(x3-1)*(x4-1);']
```

```python
>>> from phcpy.factor import solve, write_decomposition
>>> deco = solve(4, 3, pols, verbose=False)
>>> write_decomposition(deco)
```

We see the common factor $x1-1$ which defines a three dimensional solution plane. The factor $x1-2$ leads to a two dimensional solution plane, with the additional factor $x2-1$. Furthermore, the system in `pols` has twelve lines as solutions and four isolated solution points.

The first argument 4 in the `solve(4, 3, pols, verbose=False)` is the number of variables in the polynomials in `pols`. The second argument 3 equals the top dimension of the solution set. The `write_decomposition()` confirms there is one three dimensional linear component, one two dimensional linear component, twelve lines, and four isolated solutions.

### 3.3.6 diagonal homotopies

Given two witness sets, with diagonal homotopies we can compute generic points on the intersection of the algebraic sets represented by the witness sets, and thus obtain a witness set of the intersection. This section illustrates the intersection of the unit sphere with a cylinder. This intersection defines a quartic curve.

We start with equations for the unit sphere and a cylinder:

```python
>>> sph = 'x^2 + y^2 + z^2 - 1;'
>>> cyl = 'x^2 + y - y + (z - 0.5)^2 - 1;'
```

Observe the line in the assignment to `cyl`. With this trick we initialize the symbol table for the witness set computation, ensuring that $y$ is present.

Next, we compute a witness sets for the sphere and the cylinder:

```python
>>> from phcpy.sets import witness_set_of_hypersurface as witsurf
>>> sphwit = witsurf(3, sph)
>>> spheqs, sphpts = sphwit
>>> cylwit = witsurf(3, cyl)
>>> cyleqs, cylpts = cylwit
```

Once we have two witness sets, we call the `diagonal_solver` method to compute a witness set for the intersection:

```python
>>> from phcpy.diagonal import diagonal_solver as diagsolve
>>> quawit = diagsolve(3, 2, spheqs, sphpts, 2, cyleqs, cylpts)
>>> quaeqs, quapts = quawit
>>> for pol in quaeqs:
...     print(pol)
>>> for sol in quapts:
...     print(sol)
```
3.4 some interesting examples and families

PHCpack has been tested on many examples of polynomial systems taken from the research literature. The module examples exports some of those examples. Running python examples.py at the command prompt performs a regression test, solving all examples.

Polynomial systems often occur in families and are defined for any number of equations and variables.

3.4.1 interactive regression testing

An interactive use of examples.py at the Python prompt can go as follows:

```python
>>> from phcpy.examples import noon3
>>> f = noon3()
>>> for p in f: print(p)
... x1*x2^2 + x1*x3^2 - 1.1*x1 + 1;
x2*x1^2 + x2*x3^2 - 1.1*x2 + 1;
x3*x1^2 + x3*x2^2 - 1.1*x3 + 1;
```

The functions in examples.py returns the polynomials as lists of strings. If we want to solve the system defined by f, we continue the above session as

```python
>>> from phcpy.solver import solve
>>> s = solve(f,silent=True)
>>> len(s)
21
>>> print(s[0])
t : 1.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
x1 : -1.65123467890611E-01 -7.61734168646636E-01
x2 : 8.98653694263692E-01 -3.48820047576431E-01
x3 : 8.98653694263692E-01 -3.48820047576431E-01
== err : 3.034E-16 = rco : 2.761E-01 = res : 5.974E-16 =
```

The example session continues in the description of the module solutions.

3.4.2 the cyclic n-roots problem

One such noteworthy family is the cyclic n-roots problem:

```python
>>> from phcpy.families import cyclic
>>> c4 = cyclic(4)
>>> for p in c4: print(p)
... 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;
```
3.5 numerical Schubert calculus

The module schubert.py exports the hypersurface and quantum Pieri homotopies to solve the following Schubert problem: Given a sequence of generic m-planes and a corresponding sequence of interpolation points, compute all maps of degree q that meet the given m-planes nontrivially at the interpolation points.

3.5.1 Pieri homotopies

The Pieri homotopies illustrates the homotopy approach.

1. Based on the dimension of the input problem, there is a formal root count on the number of solutions, a root count that is exact for sufficiently generic instances of the input; and an upper bound for the number of isolated solution in all cases.

2. For sufficiently generic instances of the input, the performance of homotopies is optimal in the sense that every solution path defined by the homotopies ends at an actual solution of the problem.

The methods exported by the schubert module do the following:

1. Compute the formal root count for any m, p, and q. This calculation goes fast and gives an impression on the hardness of the problem.
2. Generate random instances of the problem for any (m,p,q).
3. Compute all solutions with the Pieri homotopies.
4. Verify the solutions with solutions.
5. Generate a instance of the problem known to be fully real.

The session below runs the Pieri homotopies to compute all linear maps that produce 2-planes meeting 8 given 2-planes at random interpolation points:

```python
>>> from phcpy.schubert import pieri_root_count
>>> (m,p,q) = (2,2,1)
>>> n = m*p + q*(m+p)
>>> r = pieri_root_count(m,p,q)
Pieri root count for (2, 2, 1) is 8
the localization poset :
   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)
>>> from phcpy.schubert import random_complex_matrix
>>> L = [random_complex_matrix(m+p,m) for k in range(n)]
>>> points = random_complex_matrix(n,1)
>>> from phcpy.schubert import run_pieri_homotopies
>>> (f,fsols) = run_pieri_homotopies(m,p,q,L,points)
```

The function test() of the module schubert runs an interactive session to solve instances that are fully real (in case q = 0).
3.5.2 Littlewood-Richardson homotopies

With the Littlewood-Richardson homotopies we can solve general Schubert problems. The input to a Schubert problem is a sequence of n-by-n matrices and a corresponding list of intersection conditions, represented by brackets. For example, the bracket [2, 4, 6] imposes on a 3-plane in 6-space that it meets nontrivially the space spanned by the first two columns of the corresponding matrix in a line and that it meets the space spanned by the first four columns of the corresponding matrix in a 2-plane.

For a generic sequence of input matrices, there are exactly two 3-planes in 6-space that satisfy the conditions imposes by the three brackets [2,4,6], [2,4,6], and [2,4,6], as computed in the session below.

```python
>>> from phcpy.schubert import resolve_schubert_conditions as rsc
>>> brackets = [[2,4,6],[2,4,6],[2,4,6]]
>>> rsc(6,3,brackets)
the dimension of the planes : 3
the number of conditions : 3
[2 3 4] and [2 4 6] are not happy and will not create any children.
[1 3 5] and [2 4 6] are happy and will create children...
[1 2 6] and [2 4 6] are not happy and will not create any children.
The new formal equations :
+2[1 3 5]*[2 4 6] = +2[1 2 3]
All formal equations in the intersection poset :
+2[1 3 5]*[2 4 6] = +2[1 2 3]
The intersection condition resolved :
= +2[1 2 3]
```

To compute the 2-planes, we run the Littlewood-Richardson homotopies, continuing the session from above:

```python
>>> from phcpy.schubert import littlewood_richardson_homotopies as lrh
>>> (count, flags, sys, sols) = lrh(6, 3, brackets, verbose=False)
>>> count
2
>>> for sol in sols: print(sol)
...
t : 1.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
x11 : -1.95764646993258E-01 1.07253045769427E+00
x32 : 2.69552376387238E-01 -4.99588315456159E-01
x53 : -9.21562255223665E-01 -9.28437273121748E-01
== err : 3.559E-16 = rco : 4.125E-02 = res : 7.772E-16 =
t : 1.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
x11 : -5.85142692165828E-01 -9.14161314393362E-02
x32 : -5.16006715209336E-01 3.41609194636644E-01
x53 : -6.60253695726872E-02 -1.15262273262567E+00
>>> len(sys)
13
```

The Littlewood-Richardson homotopies computed two solutions of a system of 13 equations in 3 unknowns.
3.6 Newton polytopes, monomial maps, and power series

The Newton polytopes of the polynomial system provide important information about the structure of the solution sets. The module `phcpy.polytopes` provides an interface to the convex hull methods of PHCpack. It also provides a directer interface to the mixed volume calculator, directer in the sense that the user can enter the supports directly, without having to formulate a polynomial system.

Systems that have exactly two monomials with nonzero coefficient in every equation are called binomial systems. Although such binomial systems are very particular, because of their sparse structure, they can be solved much faster. The module `phcpy.maps` provides a Python interface to the solvers of binomial systems.

The classical arithmetic can be extended to the field of truncated power series. In this field, Newton’s method computes power series solutions of polynomial systems. The module `phcpy.series` exports functions to compute power series with Newton’s method. Power series are input to Padé approximants, which lead to more accurate path trackers, exported by the module `phcpy.curves`.

3.6.1 convex hulls of lattice polytopes

The session below illustration the calculation of the convex hull of a configuration of seven points in the plane. The points are generated at random, with coordinates between -9 and +9.

```python
>>> from phcpy.polytopes import random_points as rp
>>> from phcpy.polytopes import planar_convex_hull as pch
>>> points = rp(2,7,-9,9)
>>> points
[(9, 8), (5, 6), (6, 0), (2, -5), (4, -1), (9, -4), (-1, -6)]
>>> (vertices, normals) = pch(points)
>>> vertices
[(9, 8), (5, 6), (-1, -6), (9, -4)]
>>> normals
[(1, -2), (2, -1), (-1, 5), (-1, 0)]
```

The output of the convex hull method consists of a tuple of two lists. The first list is the list of vertices. For this particular example, seven points were given on input, and only four of those points are corners of the convex hull. The list of vertices is ordered cyclically: two consecutive vertices span an edge of the polygon and the last and first vertex also span an edge as a polygon has exactly as many vertices as edges. The second list in the output is the list of inner normals, which are vectors perpendicular to the edges. Taking the inner product of the normal with the points that span an edge yields the same value for each point on the edge, and that value is minimal for all points in the polygon. For the example above for the inner normal (1, -2) and the two points (9, 8) and (5, 6), we have

\[ 9 - 2 \times 8 = 5 - 2 \times 6 = -7 \]

as the edge lies on the edge of the half plane defined by the inequality

\[ x_1 - 2x_2 \geq -7 \]

which holds for all points in the polygon spanned by the points in the example. The inner normals define the half planes that cut out the polygon.

For a convex hull of a point configuration in 3-space, consider the example in the session below:

```python
>>> from phcpy.polytopes import random_points as rp
>>> points = rp(3,10,-9,9)
>>> for point in points: print(point)
... (continues on next page)
```
(5, 9, -5)
(0, 0, 1)
(-3, -4, -1)
(-9, -3, -3)
(-5, 3, -8)
(-4, 3, 7)
(2, -3, 8)
(9, 3, -9)
(7, 4, -2)
(1, -8, 1)

```python
>>> from phcpy.polytopes import convex_hull as ch
>>> facets = ch(3, points)
computed 12 facets
>>> for facet in facets: print(facet)
... (-597, [90, -65, -6], [4, 5, 6], [1, 2, 3])
... (-84, [1, 11, 14], [4, 8, 5], [7, 11, 0])
... (-281, [30, -49, -2], [5, 1, 6], [11, 4, 0])
... (-51, [6, 5, -6], [4, 6, 7], [0, 4, 5])
... (-203, [-22, -27, -30], [6, 1, 7], [2, 9, 3])
... (-48, [5, 6, -5], [4, 7, 10], [3, 6, 7])
... (-684, [-127, 66, -29], [10, 7, 8], [5, 8, 7])
... (-150, [1, 22, 25], [4, 10, 8], [5, 6, 1])
... (-315, [-59, 15, -19], [7, 9, 8], [9, 10, 6])
... (-265, [-29, -35, -39], [7, 1, 9], [4, 10, 8])
... (-165, [-19, -10, -4], [1, 8, 9], [11, 8, 9])
... (-429, [3, -26, 42], [5, 8, 1], [1, 10, 2])
```

The output of the `convex_hull` function returns a list of facets. Each facet is represented as a tuple of four items. The first number is the value of the inner product of the vector perpendicular to the facet, given by the list in the second item of the tuple. So the first two items in the tuple define the half space defined by the facet. For the first facet, we have the inequality defined by the number -597 and the vector \([90, \ -65, \ -6]\):

\[
90x_1 - 65x_2 - 6x_3 \geq -597
\]

which holds for all points \((x_1, x_2, x_3)\) in the convex hull. The equality \(90x_1 - 65x_2 - 6x_3 = -597\) holds for all points that lie on the first facet in the list of facets above. The third item in the representation of a facet is the list of numbers to the points that span the facet. In the example above, the first facet is spanned by the points 4, 5, 6 in the input list points. Note that the counting of the points starts at one and not at zero. The last item in the representation of a facet is the list of facets that are adjacent to the facet. For the first facet, facets 1, 2, and 3 are adjacent to it. The counting of the facets starts at zero, so the first facet has label zero.

From the list of facets we can extract all vertex points. If we continue with the session from above:

```python
>>> vertices = []
>>> for facet in facets:
...     for point in facet[2]:
...         if not point in vertices:
...             vertices.append(point)
...             break
...     break
>>> vertices
[4, 5, 6, 8, 1, 7, 10, 9]
>>> len(vertices)
8
```

We have 8 vertices and 12 facets. The points the span the facets are ordered cyclically so that two consecutive points span an edge and the last and first point span also an edge. Every edge lies in the intersection of exactly two facets.
Edges of adjacent facets are ordered in opposite order. For example, facet 0 is spanned by [4, 5, 6] and its adjacent facet 1 is spanned by [4, 8, 5], with the edge shared between both of them oriented from 4 to 5 in facet 0 and from 5 to 4 in facet 1.

As the points in the configuration were generated sufficiently at random, the polytope is simplicial: every facet is spanned by exactly 3 points and has exactly 3 edges. As every edge is shared by exactly two facets we count every edge twice if we multiply the number of facets by three, so we have 36/2 = 18 edges.

### 3.6.2 mixed volumes

The mixed volume of a tuple of Newton polytopes if defined as the coefficient in the expansion of the volume of a linear combination of Newton polytopes. For example, for a 3-tuple of Newton polytopes:

\[
\vol(\lambda_1 P_1 + \lambda_2 P_2 + \lambda_3 P_3) = V(P_1, P_1, P_1)\lambda_1^3 \\
+ V(P_1, P_2, P_2)\lambda_1^2\lambda_2 \\
+ V(P_1, P_2, P_3)\lambda_1\lambda_2\lambda_3 \\
+ V(P_2, P_2, P_3)\lambda_2^3 \\
+ V(P_2, P_3, P_3)\lambda_2\lambda_3^2 \\
+ V(P_3, P_3, P_3)\lambda_3^3
\]

where \(\vol(\cdot)\) is the volume function and \(V(\cdot)\) is the mixed volume. For the tuple \((P_1, P_2, P_3)\), its mixed volume is \(V(P_1, P_2, P_3)\) in the expansion above.

The function \texttt{mixed_volume} expects two arguments. The first argument is the list of exponents of the \(\lambda\) variables in the volume expansion formula. The second argument of \texttt{mixed_volume} is a tuple of Newton polytopes. The session below illustrates the computation of the volume of one single polytope.

```python
>>> from phcpy.polytopes import random_points as rp
>>> from phcpy.polytopes import mixed_volume as mv
>>> p1 = rp(3, 5, -9, 9)
>>> print(p1)
[(3, 7, -3), (-1, 0, 8), (-6, -6, 8), (-6, 9, 4), (-3, 4, -7)]
>>> mv([3], [p1])
2107
```

The volume is normalized, so the standard unit simplex has volume one. To compute mixed volumes of two polytopes, we continue the session, generating another polytope:

```python
>>> p2 = rp(3, 5, -9, 9)
>>> mv([2, 1],(p1, p2))
3910
>>> mv([1, 2],(p1, p2))
3961
```

The \texttt{mixed_volume} function executes an Ada translation of MixedVol, ACM TOMS Algorithm 846 of 2005. This algorithm generates random floating point values to lift the points in the supports. The function \texttt{integer_mixed_cells} allows the user to specify integer lifting values as the last coordinate of the points in the supports.
3.6.3 solving binomial systems

The irreducible components of positive dimensional solution sets of binomial systems have coordinates that can be represented by maps of monomials in free independent variables. In this representation, there are as many free variables as the dimension of the solution set. The module \texttt{maps} exports a solver for binomial systems.

In the example below, we consider a simple system of two binomials in three variables:

```python
>>> f = [ 'x**2*y - z*x;', 'x**2*z - y**2*x;' ]
>>> from phcpy.maps import solve_binomials
>>> maps = solve_binomials(3,f)
>>> for map in maps: print(map)
```

In the printed maps, we recognize the twisted cubic, the x-axis, and the yz-plane as the three solution sets.

3.6.4 power series solutions

Newton’s method applies also to systems where the coefficients are truncated power series. The module \texttt{series} exports functions to compute power series solutions in double, double double, and quad double precision. The function \texttt{test()} of the \texttt{series} module provides an example.

As example, we consider the Viviani curve and intersect the curve with a moving plane. The parameter \( s \) defines the movement of the plane \( y = 0 \) to the plane \( y = 1 \), as in the setup below:

```python
>>> vivplane = ['(1-s)*y + s*(y-1);',
... 'x^2 + y^2 + z^2 - 4;','
... '(x-1)^2 + y^2 - 1;']
>>> vivs0 = vivplane + ['s;']
>>> from phcpy.solver import solve
>>> sols = solve(vivs0, silent=True)
>>> print(sols[0])
t : 1.00000000000000E+00 0.00000000000000E+00
m : 1
the solution for t :
  s : 0.00000000000000E+00 0.00000000000000E+00
  y : 0.00000000000000E+00 0.00000000000000E+00
  x : 0.00000000000000E+00 0.00000000000000E+00
  z : 2.00000000000000E+00 0.00000000000000E+00
== err : 0.000E+00 = rco : 3.186E-01 = res : 0.000E+00 =
```
3.6.5 approximating algebraic curves

Power series are the input to algorithms to construct rational approximations, also called Padé approximants.

A session can start with the tuning of the homotopy continuation parameters, as indicated below:

```python
>>> from phcpy.curves import tune_homotopy_continuation_parameters as tune
>>> tune()
Values of the HOMOTOPY CONTINUATION PARAMETERS :
1. gamma : (-0.797398052335-0.603453681844j)
2. degree of numerator of Pade approximant : 4
3. degree of denominator of Pade approximant : 4
4. maximum step size : 0.1
5. minimum step size : 1e-06
6. multiplication factor of the series step : 0.5
7. multiplication factor of the pole radius : 0.5
8. tolerance on the residual of the predictor : 0.001
9. tolerance on the residual of the corrector : 1e-8
10. tolerance on zero series coefficients : 1e-12
11. maximum number of corrector steps : 4
12. maximum steps on a path : 1000
To change a value, give an index (0 to exit) : 0
```

The function `tune()` enters an interactive loop. In this loop the user can alter all values for the twelve parameters. In the session above, the user entered zero to exit the loop without modifications to the default values of the parameters. The parameters can be queried and altered separately with the `get_` and `set_` functions of the module curves.

To illustrate the application of the trackers, we consider a small example of the Katsura family of systems.

```python
>>> from phcpy.families import katsura
>>> k3 = katsura(3)
>>> from phcpy.solver import total_degree_start_system as startsys
>>> (k3q, k3qsols) = startsys(k3)
>>> len(k3qsols)
8
```

The total degree start system for this problem of three quadrics has eight solutions. With the target system in `k3`, the start system in `k3q`, and the start solutions in `k3qsols`, we can launch the path tracker in standard double precision as follows:

```python
>>> from phcpy.curves import standard_track as track
>>> k3sols = track(k3, k3q, k3qsols, "/tmp/out", True)
```

The solutions at the end of the paths are assigned to `k3sols`. The output will be written to the file with name `/tmp/out` and the verbose flag is set to `True`. The verbose flag is optional. If no output to file is needed, then also the file name may be omitted. If an empty string is given as the value for the file name and `True` for the verbose option, then all extra output is written to screen.

To print the last 40 lines of `/tmp/out` one can do the following:

```python
>>> file = open("/tmp/out")
>>> lines = file.readlines()
>>> for k in range(-40,0): print(lines[k][:-1])
```

The module exports functions to run the path trackers step by step. In each step, the user can retrieve the power series, the Padé approximants, its poles, the step size and solution.

To test this step-by-step path tracking, do
>>> from phcpy.curves import test_next_track as test
>>> test()

The default precision is double, providing ‘dd’ or ‘qd’ to the argument of test() sets the precision respectively to double double and quad double.

## 3.7 a graphical user interface

As a programmers interface, phcpy was developed in a Terminal window, of 80 characters wide and 40 lines long. Python comes with Tkinter, which provides tools for building a graphical user interface. The goal of the module dashboard is to develop a graphical user interface to the methods of phcpy.

### 3.7.1 solving with a click of a button

A very basic graphical user interface to the blackbox solver consists of two text widgets: one for the input and another for the output; one button for the user to call the blackbox solver, and then two labels to document the functionality of the text widgets.

A screen shot of a basic interface to the blackbox solver is shown in Fig. 3.3.

The code to launch this GUI is as follows.

```python
>>> from phcpy.dashboard import launchsolver
>>> from phcpy.families import cyclic

>>> launchsolver(cyclic(5))
```

If called without arguments, as launchsolver(), then the input text widget is empty and the user must enter the polynomials in the system.

### 3.7.2 scrolling a list of solutions

The blackbox solver solve of phcpy.solver returns a list of strings. In the command line mode, we can print the solutions, one after the other. The scrollsols function launches a simple interface to scroll through the list of solutions, by clicking on previous or next buttons. The session below illustrates the scrolling through the solutions of the cyclic 5-roots problem.

```python
>>> from phcpy.families import cyclic
>>> c5 = cyclic(5)
>>> from phcpy.solver import solve
>>> sols = solve(c5, silent=True)
>>> from phcpy.dashboard import scrollsols

>>> scrollsols(sols)
```

The window that then pops up is shown in Fig. 3.4. Note that, because of different choices of random constants, the first solution will most likely be different at each run.
Fig. 3.3: Solving cyclic 5-roots with a click of a button.
3.7.3 making a coordinate plot of solutions

Solutions have coordinates in the complex plane. As in the case of the cyclic 5-roots problem, a plot of one of the coordinates in the complex plane reveals the pattern of the distribution in the roots, see Fig. 3.5. The plot appears in a canvas widget, in the GUI launched by the function `plotcoordinate(sols, idx)` where `sols` is the list of solutions and `idx` an index to a coordinate of the solutions.

3.8 the module server

The file `server` defines a simple client/server interaction to solve many random trinomials.

3.8.1 functions in the module server

The module `server.py` exports routines to send and receive lists of strings through sockets. These strings represent either polynomials or solutions as data interchanges between a client and a server. A simple illustration of the use of server is to solve many polynomial systems over a client/server network. The interactive main program starts up a multithreaded server. The server generates a list of random polynomial systems which are distributed among clients in a static scheme. The clients solve the polynomial systems and send solutions to the server.

```python
class server.ServerHandler(n, s, m, problems):
    Defines the action of the handler threads, using static workload balancing.

    run()
    Each handler accepts a connection from a client. Thread k send those problems in the list L whose index modulo the number of threads is k.

    server.client()
    The client connects to a server and solves a system.

    server.main()
    Launces one server and clients on the same computer.
```

Fig. 3.4: Scrolling the solutions of the cyclic 5-roots system.
Fig. 3.5: The first coordinate of cyclic 5-roots in the complex plane.
server.\texttt{recv\_strings} (\textit{sock, bufsize})

A list of strings is received via the socket \textit{sock} using buffer size \textit{bufsize}. First the number of strings is received, before the strings. On return is the list of strings.

server.\texttt{send\_strings} (\textit{sock, bufsize, items})

A list of strings in \textit{items} will be sent via the socket \textit{sock} using buffer size \textit{bufsize}. First the number of strings is sent, followed by the strings in \textit{items}.

server.\texttt{server\_connect} (\textit{nbclients, bufsize, portnum})

Connects a server to listen to \textit{nbclients} clients, using buffer size \textit{bufsize} and port number \textit{p}. Returns the socket to connect clients.

server.\texttt{solve\_system} (\textit{pols})

Calls the black box solver of PHCpack for valid input lists \textit{pols}. Returns the solution found.

server.\texttt{start\_server} (\textit{systems, nbclients})

The server has a list of systems in \textit{systems} for \textit{nbclients} clients to solve.

### 3.9 the module \texttt{phcpy.phcpy2c3}

The Python scripts in the package \texttt{phcpy} call the wrappers for the C interface to the Ada code in PHCpack. Below is the list of all functions exported by the shared object file \texttt{phcpy2c3.so}. The source code provides more detailed documentation.

#### 3.9.1 design of the Python to C interface

The design of \texttt{phcpy} depends on PHClib, a library of various collections of C functions, originally developed for message passing with the MPI library. This design is sketched in Fig. 3.6.

PHClib interfaces to the Ada routines through one single Ada procedure \texttt{use\_c2phc.adb}. The collection of parallel distributed memory programs (MPI2phc) using message passing (MPI) depends on PHClib. All C functions that are exported to the Python interface have their prototypes in the header file \texttt{phcpy2c.h} while the definitions in \texttt{phcpy2c3.c} call the proper routines in PHClib.

#### 3.9.2 the interface to PHCpack

The module interface collects the functions that parse the string representations for polynomials and solutions to pass their data through the C interface to the Ada code of PHCpack. The reverse operations return the string representations for polynomials and solutions as stored internally in PHCpack.

The functions exported by \texttt{phcpy.interface} concern the movement of data between Python and PHCpack. The \texttt{store\_} methods parse strings representing polynomials and solutions into the corresponding internal data structures. The corresponding \texttt{load\_} methods take the internal data structures for polynomials and solutions, stored in containers, and show their corresponding representations as Python strings. For example, consider the session

```
>>> from phcpy.interface import store_standard_system, load_standard_system
>>> store_standard_system(['x^2 - 1/3;'])
>>> load_standard_system()
['x^2 - 3.33333333333333E-01;']
```

The session above illustrates the parsing of a system one could use to approximate the square root of 1/3. With standard double precision, the 1/3 is approximated to about 15 decimal places.
3.9.3 wrappers to the C interface to PHCpack

A basic application of the primitive operations in phcpy2c3 is an interactive reading of a polynomial system. Assume the file example at /tmp/ contains a polynomial system, then we can do the following:

```python
>>> from phcpy.phcpy2c3 import py2c_syscon_read_standard_system as readsys
>>> from phcpy.phcpy2c3 import py2c_syscon_write_standard_system as writesys
>>> readsys()
Reading a polynomial system...
Is the system on a file ? (y/n/i=info) y

Reading the name of the input file.
Give a string of characters : /tmp/example
0
>>> writesys()
2
x^2+4*y^2-4;
2*y^2-x;
0
>>> from phcpy.phcpy2c3 import py2c_solve_system as solve
>>> solve(0)
ROOT COUNTS :
total degree : 4
```

(continues on next page)
general linear-product Bezout number : 4
based on the set structure :
  \{ x \ y \}\{ x \ y \}
  \{ x \ y \}\{ y \}
mixed volume : 4
stable mixed volume : 4

```python
>>> from phcpy.phcpy2c3 import py2c_solcon_write_standard_solutions as writesols
>>> writesols()
4 2
```

<table>
<thead>
<tr>
<th>Solution</th>
<th>t</th>
<th>m</th>
<th>the solution for t</th>
<th>x</th>
<th>y</th>
<th>err</th>
<th>rco</th>
<th>res</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00000000000000E+00</td>
<td>1</td>
<td>x : 1.23606797749979E+00</td>
<td>-9.91383530201425E-119</td>
<td>y : 7.86151377757423E-01</td>
<td>4.95691765100713E-119</td>
<td>1.567E-16</td>
<td>3.067E-01</td>
</tr>
<tr>
<td>2</td>
<td>1.00000000000000E+00</td>
<td>1</td>
<td>x : 1.23606797749979E+00</td>
<td>-9.91383530201425E-119</td>
<td>y : -7.86151377757423E-01</td>
<td>-4.95691765100713E-119</td>
<td>1.567E-16</td>
<td>3.067E-01</td>
</tr>
<tr>
<td>3</td>
<td>1.00000000000000E+00</td>
<td>1</td>
<td>x : -3.23606797749979E+00</td>
<td>3.1724272964456E-117</td>
<td>y : -1.58621364832228E-117</td>
<td>1.27201964951407E+00</td>
<td>3.703E-16</td>
<td>1.515E-01</td>
</tr>
<tr>
<td>4</td>
<td>1.00000000000000E+00</td>
<td>1</td>
<td>x : -3.23606797749979E+00</td>
<td>3.1724272964456E-117</td>
<td>y : 1.58621364832228E-117</td>
<td>-1.27201964951407E+00</td>
<td>3.703E-16</td>
<td>1.515E-01</td>
</tr>
</tbody>
</table>

With these primitive operations in phcpy2c3 we can bypass the writing and the parsing to strings.
3.10 code snippets

One extension of a Jupyter notebook is the definition of code snippets. The snippets provide examples to demonstrate the capabilities of phcpy. The titles of the sections below are the titles of the snippet menus in the notebook extension.

3.10.1 blackbox solver

1. solving trinomials
   1. solving a random case

   ```python
   from phcpy.solver import random_trinomials
   f = random_trinomials()
   for pol in f: print(pol)
   from phcpy.solver import solve
   sols = solve(f, verbose=False)
   for sol in sols: print(sol)
   print(len(sols), "solutions found")
   ```

   2. solving a specific case

   ```python
   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)
   ```

2. representations of isolated solutions

   1. from string to dictionary

   ```python
   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])
   ```

   2. verify by evaluation

   ```python
   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)
   ```

3. making a solution

```python
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)
```
4. filtering solution lists

```python
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)
```

5. coordinates, names and values

```python
from phcpy.solver import solve
p = ['x^2*y^2 + x + 1;', 'x^2*y^2 + y + 1;']
s = solve(p)
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)
```

3. reproducible runs with fixed seeds

1. fixing and retrieving the seed

```python
from phcpy.phcpy2c2 import py2c_set_seed
py2c_set_seed(2013)
from phcpy.phcpy2c2 import py2c_get_seed
print(py2c_get_seed())
```

4. shared memory parallelism

1. solving with 4 tasks

```python
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 time import time
starttime = time()
sols = solve(pols, verbose=False)
stop = time()
candidate = stop - start
print('solving with no multitasking :')
print('elapsed time : %.2f seconds' % candidate)
starttime = time()
sols = solve(pols, verbose=False, tasks=nbrt)
stop = time()
candidate = stop - start
print('solving with %d tasks :' % nbrt)
print('elapsed time : %.2f seconds' % candidate)
```

5. root counting methods

1. four different root counts

```python
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 time import time
starttime = time()
sols = solve(pols, verbose=False)
stop = time()
candidate = stop - start
print('solving with no multitasking :')
print('elapsed time : %.2f seconds' % candidate)
starttime = time()
sols = solve(pols, verbose=False, tasks=nbrt)
stop = time()
candidate = stop - start
print('solving with %d tasks :' % nbrt)
print('elapsed time : %.2f seconds' % candidate)
```
f = ['x^3*y^2 + x*y^2 + x^2;', 'x^5 + x^2*y^3 + y^2;']
from phcpy.solver import total_degree
print('the total degree :', total_degree(f))
from phcpy.solver import m_homogeneous_bezout_number as mbz
(bz, part) = mbz(f)
print('a multihomogeneous Bezout number :', bz)
from phcpy.solver import linear_product_root_count as lrc
print('a linear-product root count :', lrc(f, verbose=False))
from phcpy.solver import mixed_volume
(mv, smv) = mixed_volume(f, stable=True)
print('the mixed volume :', mv)
print('the stable mixed volume :', smv)

6. Newton’s method and deflation

1. the Griewank-Osborne example

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.solver import newton_step
s2 = newton_step(p, [s])
print(s2[0])
s3 = newton_step(p, s2)
print(s3[0])
from phcpy.solver import standard_deflate
sd = standard_deflate(p, [s])
print(sd[0])

2. deflating an overconstrained system

from phcpy.solutions import make_solution
from phcpy.solver import standard_deflate
sol = make_solution(['x', 'y'], [float(1.0e-6), float(1.0e-6)])
print(sol)
pols = ['x**2;', 'x*y;', 'y**2;']
sols = standard_deflate(pols, [sol], tolrnk=1.0e-8)
print(sols[0])
sols = standard_deflate(pols, [sol], tolrnk=1.0e-4)
print(sols[0])

7. equation and variable scaling

1. 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, verbose=False)
print(psols[0])

2. solving after scaling

p = ['0.000001*x^2 + 0.000004*y^2 - 4;', '0.000002*y^2 - 0.001*x;']
from phcpy.solver import standard_scale_system as scalesys
from phcpy.solver import standard_scale_solutions as scalesols
(q, c) = scalesys(p)
for pol in q: print(pol)
qsols = solve(q, verbose=False)
ssols = scalesols(len(q), qsols, c)
for sol in ssols: print(sol)

### 3.10.2 path trackers

#### 1. a simple example

1. a total degree start system

```python
from phcpy.solver import total_degree
from phcpy.solver import total_degree_start_system
from phcpy.trackers import 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 = track(p, q, qsols)
for sol in s: print(sol)
```

2. track one solution path

```python
from phcpy.solver import total_degree_start_system
from phcpy.trackers import track
p = ['x^2 + 4*y^2 - 4;', '2*y^2 - x;']
(q, qsols) = total_degree_start_system(p)
s1 = track(p, q, [qsols[2]])
print(s1[0])
s2 = track(p, q,[qsols[2]])
print(s2[0])
```

2. fixing the gamma constant

1. specifying the gamma parameter

```python
from phcpy.solver import total_degree_start_system
from phcpy.trackers import track
p = ['x^2 + 4*y^2 - 4;', '2*y^2 - x;']
(q, qsols) = total_degree_start_system(p)
s3 = track(p, q, [qsols[2]], gamma=complex(0.8243728063190.56604723848934))
print('the solution at the end: ')
print(s3[0])
```

3. give the next solution on a path

1. tracking with a generator

```python
from phcpy.solver 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_standard_tracker
from phcpy.trackers import initialize_standard_solution
from phcpy.trackers import next_standard_solution
```

(continues on next page)
initialize_standard_tracker(p, q)
initialize_standard_solution(len(p), s[0])
s1 = next_standard_solution()
print('the next point on the solution path :')
print(s1)
print(next_standard_solution())
print(next_standard_solution())
initialize_standard_solution(len(p), s[1])
points = [next_standard_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]
for x in xvals: print(x)

2. plotting trajectories

p = ['x^2 + y - 3;', 'x + 0.125*y^2 - 1.5;']
print('constructing a total degree start system ...')
from phcpy.solver import total_degree_start_system as tds
q, qsols = tds(p)
print('number of start solutions :', len(qsols))
from phcpy.trackers import initialize_standard_tracker
from phcpy.trackers import initialize_standard_solution
from phcpy.trackers import next_standard_solution
initialize_standard_tracker(p, q, False)
from phcpy.solutions import strsol2dict
import matplotlib.pyplot as plt
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_standard_solution(len(p), startsol)
dictsol = strsol2dict(startsol)
xpoints = [dictsol['x']]
ypoints = [dictsol['y']]
for k in range(300):
    ns = next_standard_solution()
dictsol = strsol2dict(ns)
xpoints.append(dictsol['x'])
ypoints.append(dictsol['y'])
tval = eval(dictsol['t'].lstrip().split(' ')[0])
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()

4. solving with polyhedral homotopies
   1. solving a random coefficient system

   \[ p = ['x^3*y^2 - 3*x^3 + 7;','x*y^3 + 6*y^3 - 9;'] \]
   
   from phcpy.solver import mixed_volume
   print('the mixed volume :', mixed_volume(p))
   from phcpy.solver import random_coefficient_system
   (q, qsols) = random_coefficient_system(\text{verbose=False})
   print('the number of start solutions :', len(qsols))
   from phcpy.trackers import track
   psols = track(p, q, qsols)
   print('the number of solutions at the end :', len(psols))
   for sol in psols: print(sol)

5. Newton’s method at higher precision
   1. using a linear-product start system

   \[ p = ['x*y^3 + y - 2;', 'x^3*y + x - 8;'] \]
   
   from phcpy.solver import linear_product_root_count
   r = linear_product_root_count(p)
   from phcpy.solver import random_linear_product_system
   (q, qsols) = random_linear_product_system(p)
   print('the number of start solutions :', len(qsols))
   from phcpy.trackers import track
   psols = track(p, q, qsols)
   print('the number of end solutions :', len(psols))
   from phcpy.solver import newton_step
   psols_dd = newton_step(p, psols, \text{precision='dd'})
   print('the solutions in double double precision :')
   for sol in psols_dd: print(sol)

6. multitasked path tracking
   1. tracking with 4 tasks

   from phcpy.solver import random_linear_product_system as rlps
   from phcpy.families import noon
   from phcpy.trackers import track
   nbrt = 4 \# number of tasks
   pols = noon(5)
   print('solving the 5-variable Noonburg system :')
   for pol in pols: print(pol)
   (startpols, startsols) = rlps(pols)
   print('number of paths :', len(startsols))
   from time import time, "starttime = time()"
   sols = track(pols, startpols, startsols)
   stoptime = time()
   elapsed = stoptime - starttime
   print('elapsed time with no multitasking : %2f seconds' \% elapsed)
   starttime = time()
   sols = track(pols, startpols, startsols, tasks=nbrt)
   stoptime = time()
elapsed = stoptime - starttime
print('elapsed time with %d tasks : %.2f seconds' % (nbrt, elapsed))

7. sweep homotopies
   1. towards a quadratic turning point
      
      circle = ['x^2 + y^2 - 1;', 'y*(1-s) + (y-2)*s;']
      from phcpy.solutions import make_solution as makesol
      first = makesol(['x', 'y', 's'], [float(1), float(0), float(0)])
      second = makesol(['x', 'y', 's'], [float(-1), float(0), float(0)])
      start = [first, second]
      from phcpy.sweepers import standard_real_sweep as sweep
      newsols = sweep(circle, start)
      print(newsols[0])

8. real versus complex sweeps
   1. complex parameter homotopy continuation
      
      circle = ['x^2 + y^2 - 1;']
      from phcpy.solutions import make_solution as makesol
      first = makesol(['x', 'y'], [float(1), float(0)])
      second = makesol(['x', 'y'], [float(-1), float(0)])
      start = [first, second]
      par = ['y']
      start = [0, 0]
      target = [2, 0]
      from phcpy.sweepers import standard_complex_sweep as sweep
      newsols = sweep(circle, start, 2, par, start, target)
      print(newsols[0])

9. a polyhedral end game
   1. numerical tropism computation
      
      from phcpy.tuning import order_endgame_extrapolator_set as set
      from phcpy.tuning import order_endgame_extrapolator_get as get
      f = ['x + y^3 - 1;', 'x + y^3 + 1;']
      from phcpy.solver import mixed_volume as mv
      from phcpy.solver import random_coefficient_system as rcs
      print('the mixed volume :', mv(f))
      (g, gsols) = rcs(f)
      print('the number of start solutions :', len(gsols))
      from phcpy.trackers import standard_double_track as track
      sols = track(f, g, gsols)
      from phcpy.tropisms import standard_retrieve as retrieve
      (w, d, e) = retrieve(len(sols), len(f))
      print('the numerical direction :', d)
      print('the error :', e)
      print(w)
3.10.3 solution sets

1. witness sets

1. embedding the twisted cubic

```python
twisted = ['x^2 - y;', 'x^3 - z;']
from phcpy.sets import embed
e = embed(3,1,twisted)
for pol in e: print(pol)
```

2. a witness set for the twisted cubic

```python
twisted = ['x^2 - y;', 'x^3 - z;']
from phcpy.sets import embed
e = embed(3,1,twisted)
from phcpy.solver import solve
s = solve(e, verbose=False)
print('number of generic points :', len(s))
for sol in s: print(sol)
```

2. homotopy membership test

1. cyclic 4-roots on coordinates

```python
from phcpy.families import cyclic
c4 = cyclic(4)
from phcpy.sets import embed
c4e1 = 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')
for sol in genpts: print(sol)
point0 = [1, 0, -1, 0, 1, 0, -1, 0]
from phcpy.sets import membertest
print('point0 :', point0)
print('Is point0 a member ?', membertest(c4e1, genpts, 1, point0))
point1 = [1, 0, 1, 0, -1, 0, -1, 0]
print('point1 :', point1)
print('Is point1 a member ?', membertest(c4e1, genpts, 1, point1))
```

2. cyclic 4-roots on solutions

```python
from phcpy.families import cyclic
c4 = cyclic(4)
from phcpy.sets import embed
c4e1 = 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')
for sol in genpts: print(sol)
names = ['x0', 'x1', 'x2', 'x3']
coord0 = [complex(1, 0), complex(-1, 0), complex(1, 0), complex(-1, 0)]
from phcpy.solutions import make_solution
point0 = make_solution(names, coord0)
from phcpy.sets import is_member
print('point0 :', point0)
print('Is point0 a member ?', is_member(point0, genpts))
```
3. cascade of homotopies
   1. an illustrative example

   ```python
   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 run_cascade
   otp = run_cascade(3, 2, pols)
   dims = otp.keys()
   dims.sort(reverse=True)
   for dim in dims: print('number of solutions at dimension', dim, ':', len(otp[dim][1]))
   ```

2. a Laurent system

   ```python
   pol1 = '0.710358341606049*t1 + 0.46*t2 - 0.41*t3 + 0.24076130555115 + 1.07248215701824*I;'  
   pol2 = 't2*(-0.11 + 0.49*I) + 0.41*t3 - 0.502195181179589*t4 + 0.41*t5;'  
   pol3 = '0.502195181179589*t4 + t5*(-0.0980434782608696 + 0.4367391304434783*I) - 0.775518556636566*t6 - 1.2;'  
   pol4 = '0.710358341606049*t1**(-1) + 0.46*t2**(-1) - 0.41*t3**(-1) + 0.24076130555115 - 1.07248215701824*I;'  
   pol5 = 't2**(-1)*(-0.11 - 0.49*I) + 0.41*t3**(-1) - 0.502195181179589*t4**(-1) + 0.41*t5**(-1);'  
   pol6 = '0.502195181179589*t4**(-1) + t5**(-1)*(-0.0980434782608696 - 0.4367391304434783*I) - 0.775518556636566*t6**(-1) - 1.2;'  
   pols = [pol1, pol2, pol3, pol4, pol5, pol6]
   from phcpy.cascades import run_cascade
   otp = run_cascade(6, 1, pols, islaurent=True)
   (epols, esols) = otp[1]
   print('the generic points at a 1-dimensional curve :')
   for sol in esols: print(sol)
   ```

4. factoring into irreducibles
   1. factoring a cubic polynomial

   ```python
   p = '(x+1)*(x^2 + y^2 + 1);'
   from phcpy.sets import witness_set_of_hypersurface as wh
   (w, s) = wh(2, p)
   print('number of witness points :', len(s))
   from phcpy.factor import factor
   f = factor(1, w, s)
   for fact in f: print(fact)
   ```

5. numerical irreducible decomposition
1. an example

```python
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.factor import solve, write_decomposition
deco = solve(4, 3, pols, verbose=False)
write_decomposition(deco)
```

6. diagonal homotopies

1. sphere intersected with a cylinder

```python
sph = 'x^2 + y^2 + z^2 - 1;

cyl = 'x^2 + y - y + (z - 0.5)^2 - 1;

from phcpy.sets import witness_set_of_hypersurface as witsurf
sphwit = witsurf(3, sph)
spheqs, sphpts = sphwit
cylwit = witsurf(3, cyl)
cyleqs, cylpts = cylwit

from phcpy.diagonal import diagonal_solver as diagsolve
quawit = diagsolve(3, 2, spheqs, sphpts, 2, cyleqs, cylpts, verbose=False)
quaeqs, quapts = quawit
for pol in quaeqs: print(pol)
for sol in quapts: print(sol)
```

3.10.4 families of systems

1. systems in a paper by Noonburg

1. for linear-product start systems

```python
from phcpy.examples import noon3
f = noon3()
for p in f: print(p)
from phcpy.solver import solve
sols = solve(f)
print('the number of solutions :', len(sols))
```

2. the cyclic n-roots problem

1. for polyhedral homotopies

```python
from phcpy.families import cyclic
c5 = cyclic(5)
for p in c5: print(p)
from phcpy.solver import solve
sols = solve(c5)
print('the number of solutions :', len(sols))
```
3.10.5 Schubert calculus

1. Pieri homotopies
   1. curves in the Grassmannian

```python
from phcpy.schubert import pieri_root_count, random_complex_matrix
(m, p, q) = (2, 2, 1), "n = m*p + q*(m+p)
r = pieri_root_count(m, p, q)
L = [random_complex_matrix(m+p, m) for k in range(n)]
points = random_complex_matrix(n, 1)
from phcpy.schubert import run_pieri_homotopies
(f, fsols) = run_pieri_homotopies(m, p, q, L, points)
print('number of solutions :', len(fsols))
```

2. Littlewood-Richardson homotopies
   1. resolving Schubert conditions

```python
from phcpy.schubert import resolve_schubert_conditions as rsc
brackets = [[2, 4, 6], [2, 4, 6], [2, 4, 6]]
rsc(6, 3, brackets)
```

   2. solving a generic instance

```python
brackets = [[2, 4, 6], [2, 4, 6], [2, 4, 6]]
from phcpy.schubert import 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))
```

3.10.6 Newton polytopes

1. convex hulls of lattice polytopes
   1. vertices and edge normals

```python
from phcpy.polytopes import random_points as rp
from phcpy.polytopes import planar_convex_hull as pch
points = rp(2, 7, -9, 9)
for point in points: print(point)
(vertices, normals) = pch(points)
print('the vertex points :', vertices)
print('the edge normals :', normals)
```

   2. facets in 3-space

```python
from phcpy.polytopes import random_points as rp
points = rp(3, 10, -9, 9)
for point in points: print(point)
from phcpy.polytopes import convex_hull as ch
facets = ch(3, points)
for facet in facets: print(facet)
```

2. mixed volumes
   1. volume of one random polytope
from phcpy.polytopes import random_points as rp
from phcpy.polytopes import mixed_volume as mv
p1 = rp(3, 5, -9, 9)
print(p1)
mv([3], [p1])

2. mixed volume of two random polytopes

from phcpy.polytopes import random_points as rp
from phcpy.polytopes import mixed_volume as mv
p1 = rp(3, 5, -9, 9); p2 = rp(3, 5, -9, 9)
mv([2, 1],[p1, p2])
mv([1, 2],[p1, p2])

3. solving binomial systems
   1. solution curves are maps

f = [ 'x**2*y - z*x;', 'x**2*z - y**2*x;' ]
from phcpy.maps import solve_binomials
maps = solve_binomials(3, f)
for map in maps: print(map)

4. power series solutions
   1. intersecting the Viviani curve

plane = '(1-s)*y + s*(y-1);'
vp0 = 'x'^2 + y'^2 + z'^2 - 4;'
vp1 = '(x-1)^2 + y'^2 - 1;'
vivplane = [plane, vp0, vp1]
vivs0 = vivplane + ['s;']
from phcpy.solver import solve
sols = solve(vivs0, verbose=False)
for sol in sols: print(sol)
from phcpy.series import standard_newton_series
sersols = standard_newton_series(vivplane, sols, verbose=False)
for srs in sersols: print(srs)

3.10.7 the extension module
   1. the module interface
      1. storing and loading a system

from phcpy.interface import store_standard_system, load_standard_system
store_standard_system(["x'^2 - 1/3;"])
load_standard_system()"

2. wrappers to the C interface
   1. reading and writing a system

from phcpy.phcpy2c2 import py2c_syscon_read_standard_system as readsys
readsys()
from phcpy.phcpy2c2 import py2c_syscon_write_standard_system as writesys
writesys()

3.10. code snippets
This chapter contains the documentation of the modules in the package, mostly automatically generated from the
documentation strings in the module and of the functions exported by each module. The order of the sections in this
chapter follows the order of the previous chapters. The first section on the function in the solver module corresponds
with chapter 3, on a blackbox solver. The section headings correspond to the earlier chapter headings.

4.1 a blackbox solver for isolated solutions

The two most essential modules to solve polynomial systems are the solver module, which exports the blackbox
solver, and the solutions module, to parse the computed solutions.

4.1.1 functions in the module solver

The documentation strings of the functions exported by the module solver of the package phcpy are listed below.
The main functionality of PHCpack is its blackbox solver and the wide variety of start systems and homotopies. The
blackbox solver is exported by the function solve. An important task of the solver is the mixed-volume computation,
available in the function mixed_volume. For start systems based on the degrees of the polynomials, we have the plain
total degree, m-homogeneous Bezout numbers, and general linear-product start systems.

\[ \text{solver.dobldobl deflate} (\text{system, solutions, maxitr=3, maxdef=3, tollerr=1e-08, tolres=1e-08, tolrnk=1e-06}) \]

The deflation method augments the given system with derivatives to restore the quadratic convergence of New-
ton’s method at isolated singular solutions, in double double precision. The numerical parameters are

- \( \text{maxitr} \): the maximum number of iterations per root,
- \( \text{maxdef} \): the maximum number of deflations per root,
- \( \text{tollerr} \): tolerance on the forward error on each root,
- \( \text{tolres} \): tolerance on the backward error on each root,
- \( \text{tolrnk} \): tolerance on the numerical rank of the Jacobian matrices.
After application of deflation, the new approximations are returned.

**solver.dobldobl_linear_reduction**(pols, diagonalize=True)

Applies row reduction in double double precision on the coefficient matrix of the polynomials in the list pols. As the monomials are sorted in the total degree order, the total degree of the system may decrease as a result. If diagonalize, then the coefficient matrix will be made diagonal. On return is the list of reduced polynomials.

**solver.dobldobl_multiplicity**(system, solution, order=5, tol=1e-08, verbose=False)

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,
- **verbose**: if extra output is needed.

On return is the computed multiplicity.

**solver.dobldobl_random_coefficient_system**(verbose=True)

Runs the polyhedral homotopies and returns a random coefficient system based on the contents of the cell container, in double double precision arithmetic. For this to work, the function `mixed_volume()` must be called first.

**solver.dobldobl_random_system**(neq, nvr, nbrmon, deg, cff)

Returns a random polynomial system with coefficients in double double precision, based on the following:

- **neq**: number of equations,
- **nvr**: number of variables,
- **nbrmon**: maximum number of monomials per equation,

  - if 0, then the generated polynomials are dense,
- **deg**: upper bound on the degree of the monomials,
- **cff**: type of coefficients, must be 0, 1, or 2,

  - if 0, then random complex numbers on the unit circle,
  - if 1, then coefficients are one (or integer multiples of one),
  - if 2, then coefficients are floats in [-1,+1].

**solver.dobldobl_scale_solutions**(nvar, sols, cffs)

Scales the solutions in the list sols using the coefficients in cffs, using double double precision arithmetic. The number of variables is given in the parameter nvar. If the sols are the solutions of the polynomials in the output of doblodobl_scale_system(pols), then the solutions on return will be solutions of the original polynomials in the list pols.

**solver.dobldobl_scale_system**(pols)

Applies equation and variable scaling in double double precision to the polynomials in the list pols. On return is the list of scaled polynomials and the scaling coefficients.

**solver.dobldobl_solve**(pols, verbose=True, tasks=0, dictionary_output=False, verbose_level=0)

Calls the blackbox solver to compute all isolated solutions in double double precision. On input in pols is a list of strings. By default, the solver will print to screen the computed root counts. To make the solver silent, set the flag verbose to False. The number of tasks for multithreading is given by tasks. The default zero value for tasks indicates no multithreading. If dictionary_output, then on return is a list of dictionaries, else the returned list is a list of strings. If verbose_level is larger than 0, then the names of the procedures called in the running of the blackbox solver will be listed. The solving happens in double double precision arithmetic.
solver.dobldobl_usolve(\textit{pol, mxi, eps})

Applies the method of Durand-Kerner (aka Weierstrass) to the polynomial in the string \textit{pol}, in double double precision. The maximum number of iterations is in \textit{mxi}, the requirement on the accuracy in \textit{eps}.

solver.is_square(\textit{pols})

Given in the list \textit{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.

solver.linear_product_root_count(\textit{pols, verbose=True})

Given in \textit{pols} a list of string representations of polynomials, returns a linear-product root count based on a supporting set structure of the polynomials in \textit{pols}. This root count is an upper bound for the number of isolated solutions.

solver.linear_reduce(\textit{pols, diagonalize=True, precision='d'})

Applies row reduction to the coefficient matrix of the polynomials in the list \textit{pols}. As the monomials are sorted by total degree, the Bezout bound may decrease as a result of this row reduction. By default, if \textit{diagonalize}, the coefficient matrix will be made diagonal. The default precision is double precision. Other available precisions are double double (‘dd’) and quad double (‘qd’).

solver.m_homogeneous_bezout_number(\textit{pols})

Given in \textit{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.

solver.m_homogeneous_start_system(\textit{pols, partition, checkin=True})

For an m-homogeneous Bezout number of a polynomial system defined by a \textit{partition} of the set of unknowns, one can define a linear-product system that has exactly as many regular solutions as the Bezout 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 \textit{pols} and the partition. If \textit{checkin}, then the list \textit{pols} is tested to see if \textit{pols} defines a square polynomial system. If the input system is not square, then an error message is printed and None is returned.

solver.m_partition_bezout_number(\textit{pols, partition})

There are as many m-homogeneous Bezout numbers as there are partitions of the set of unknowns of a polynomial system. Given in \textit{pols} the string representations of a polynomial system in as many variables as equations, and a string representation of a \textit{partition} of the set of unknowns, this function returns the m-homogeneous Bezout number corresponding to the given partition.

solver.mixed_volume(\textit{pols, stable=False, checkin=True})

Given in \textit{pols} a list of string representations of polynomials, this function returns the mixed volume of the system. This is an interface to Algorithm 846: MixedVol of ACM TOMS, developed by Tangan Gao, T.Y. Li, Mengnien Wu, and Li Xing. If the option \textit{stable} is set to True, then on return is a tuple containing the mixed volume and the stable mixed volume. The mixed volume counts the solutions with all their coordinates nonzero, the stable mixed volume counts all affine roots. Note that the stable mixed volume does not apply to systems with negative exponents. Incorrectly parsed strings will result in a negative value on return. If \textit{checkin}, then the system is test for being square and if then the system is not square, then an error message is printed and -1 is returned.

solver.mixed_volume_by_demics(\textit{pols, stable=False, checkin=True})

Given in \textit{pols} a list of string representations of polynomials, DEMiCs is called to compute the mixed volume of the Newton polytopes spanned by the supports of the polynomials in the system. DEMiCs applies dynamic enumeration to compute all mixed cells, was developed by Tomohiko Mizutani, Akiko Takeda, and Masakazu Kojima. If the option \textit{stable} is set to True, then on return is a tuple containing the mixed volume and the stable mixed volume. The mixed volume counts the solutions with all their coordinates nonzero, the stable mixed volume counts all affine roots. Note that the stable mixed volume does not apply to systems with negative exponents.
exponents. If checkin, then the system will be checked for being square and if then the system is not square, an
error message is printed and -1 is returned.

solver.multprec_usolve(pol, mxi, eps, decimals)
Applies the method of Durand-Kerner (aka Weierstrass) to the polynomial in the string pol, in arbitrary multi-
precision, the number of decimal places in the precision is in decimals. The maximum number of iterations is
in mxi, the requirement on the accuracy in eps.

solver.names_of_variables(pols)
Returns a list of strings with the names of all variables that occur in the list of polynomials (given as strings) in
pols.

solver.newton_laurent_step(system, solutions, precision='d', decimals=100)
Applies one Newton step to the solutions of the Laurent system. For each solution, prints its last line of diag-
nostics. Four 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)),
- mp: arbitrary precision, where the number of decimal places in the working precision is determined by decimals.

solver.newton_step(system, solutions, precision='d', decimals=100)
Applies one Newton step to the solutions of the system. For each solution, prints its last line of diagnostics. Four
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)),
- mp: arbitrary precision, where the number of decimal places in the working precision is determined by decimals.

solver.newton_steps(system, solutions, accuracy=8, maxsteps=4, maxprec=256)
Runs a sequence of variable precision Newton steps to approximate solutions accurate up to a specified number
of decimal places. In addition to the system and solutions, there are three parameters:

- accuracy: number of decimal places wanted to be accurate,
- maxsteps: maximum number of Newton steps,
- maxprec: maximum number of decimal places in the precision used to estimate the condition numbers.

solver.number_of_symbols(pols)
Returns the number of symbols used as variables in the polynomials in the list pols. This function helps to
determine whether a system is square (that is: has as many equations as unknowns) or not.

solver.permute_dobldobl_system(pols)
Permutes the equations in the list of polynomials in pols with coefficients in double double precision, along the
permutation used in the mixed volume computation.

solver.permute_quaddobl_system(pols)
Permutes the equations in the list of polynomials in pols with coefficients in quad double precision, along the
permutation used in the mixed volume computation.

solver.permute_standard_system(pols)
Permutes the equations in the list of polynomials in pols with coefficients in standard double precision, along
the permutation used in the mixed volume computation.
The deflation method augments the given system with derivatives to restore the quadratic convergence of Newton’s method at isolated singular solutions, in quad double precision. The numerical parameters are

- `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.

After application of deflation, the new approximations are returned.

The deflation method augments the given system with derivatives to restore the quadratic convergence of Newton’s method at isolated singular solutions, in quad double precision. The numerical parameters are

- `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.

After application of deflation, the new approximations are returned.

The deflation method augments the given system with derivatives to restore the quadratic convergence of Newton’s method at isolated singular solutions, in quad double precision. The numerical parameters are

- `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.

After application of deflation, the new approximations are returned.

The deflation method augments the given system with derivatives to restore the quadratic convergence of Newton’s method at isolated singular solutions, in quad double precision. The numerical parameters are

- `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.

After application of deflation, the new approximations are returned.

The deflation method augments the given system with derivatives to restore the quadratic convergence of Newton’s method at isolated singular solutions, in quad double precision. The numerical parameters are

- `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.

After application of deflation, the new approximations are returned.
**solver.quaddobl_scale_system** *(pols)*
Applies equation and variable scaling in quad double precision to the polynomials in the list *pols*. On return is
the list of scaled polynomials and the scaling coefficients.

**solver.quaddobl_solve** *(pols, verbose=True, tasks=0, dictionary_output=False, verbose_level=0)*
Calls the blackbox solver to compute all isolated solutions in quad double precision. On input in *pols* is a list
of strings. By default, the solver will print to screen the computed root counts. To make the solver silent, set
the flag *verbose* to False. The number of tasks for multithreading is given by *tasks*. The zero value for *tasks*
indicates no multithreading. If *dictionary_output*, then on return is a list of dictionaries, else the returned list is
a list of strings. If *verbose_level* is larger than 0, then the names of the procedures called in the running of the
blackbox solver will be listed. The solving happens in quad double precision arithmetic.

**solver.quaddobl_usolve** *(pol, mxi, eps)*
Applies the method of Durand-Kerner (aka Weierstrass) to the polynomial in the string *pol*, in quad double
precision The maximum number of iterations is in *mxi*, the requirement on the accuracy in *eps*.

**solver.random_coefficient_system** *(verbose=True, precision='d' )*
Runs the polyhedral homotopies and returns a random coefficient system based on the contents of the cell
container. For this to work, the function *mixed_volume()* must be called first. 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)).

**solver.random_linear_product_system** *(pols, tosolve=True, checkin=True)*
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.

**solver.random_system** *(neq, nvr, nbrmon, deg, cff, precision='d' )*
Generates a random polynomial system based on the following:

- *neq*: number of equations,
- *nvr*: number of variables,
- *nbrmon*: maximum number of monomials per equation,
  - if 0, then the generated polynomials are dense,
- *deg*: upper bound on the degree of the monomials,
- *cff*: type of coefficients, must be 0, 1, or 2,
  - if 0, then random complex numbers on the unit circle,
  - if 1, then coefficients are one (or integer multiples of one),
  - if 2, then coefficients are floats in [-1,+1],
- *precision*: the precision of the coefficients,
  - if ‘d’, the precision of the coefficients is double,
  - if ‘dd’, the precision of the coefficients is double double,
  - if ‘qd’, the precision of the coefficients is quad double.

**solver.random_trinomials()**
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)
On input in sys are two random trinomials 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, verbose=True, tasks=0, myfocus=0, precision='d', checkin=True, dictionary_output=False, verbose_level=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, use solve in phcpy.factor. On input in pols is a list of strings. By default, the solver will print to screen the computed root counts. To make the solver silent, set the flag verbose to False. The number of tasks for multithreading is given by tasks. The default zero value for tasks indicates no multithreading. If the myfocus 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 verbose_level 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)
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.standard_condition_report(infilename, outfilename, maxit=4, tolres=1e-08, tolderr=1e-08, tolsing=1e-08, verbose=True)
Computes a condition report for the system and the solutions on the file with name infile, in double precision. This report is intended for huge solution lists and the solutions are not exported into the Python session.

solver.standard_deflate(system, solutions, maxitr=3, maxdef=3, tolerr=1e-08, tolres=1e-08, tolrnk=1e-06)
The deflation method augments the given system with derivatives to restore the quadratic convergence of Newton’s method at isolated singular solutions, in standard double precision. The numerical parameters are

- **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.

After application of deflation, the new approximations are returned.

solver.standard_linear_reduction(pols, diagonalize=True)
Applies row reduction in standard double precision on the coefficient matrix of the polynomials in the list pols. As the monomials are sorted in the total degree order, the total degree of the system may decrease as a result. If diagonalize, then the coefficient matrix will be made diagonal. On return is the list of reduced polynomials.

solver.standard_multiplicity(system, solution, order=5, tol=1e-08, verbose=False)
Computes the multiplicity structure in standard 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,
verbose: if extra output is needed.

On return is the computed multiplicity.

**solver.standard_nonlinear_reduction**(pols, eqmax=100, spmax=100, rpmax=100, verbose=True)

Applies nonlinear reduction in standard double precision on the polynomials in the list `pols`. In addition to `pols`, three integers are part of the input: `eqmax` is the maximum number of equal degree replacements, `spmax` is the maximum number of computed S-polynomials, and `rpmax` is the maximum number of computed R-polynomials. By default, `verbose` is True and the counts of equal degree replacements, computed S-polynomials and R-polynomials are written. On return is the list of reduced polynomials.

**solver.standard_random_coefficient_system**(verbose=True)

Runs the polyhedral homotopies and returns a random coefficient system based on the contents of the cell container, in standard double precision arithmetic. For this to work, the function `mixed_volume()` must be called first.

**solver.standard_random_system**(neq, nvr, nbrmon, deg, cff)

Returns a random polynomial system with coefficients in standard double precision, based on the following:

- **neq**: number of equations,
- **nvr**: number of variables,
- **nbrmon**: maximum number of monomials per equation,
  - if 0, then the generated polynomials are dense,
- **deg**: upper bound on the degree of the monomials,
- **cff**: type of coefficients, must be 0, 1, or 2,
  - if 0, then random complex numbers on the unit circle,
  - if 1, then coefficients are one (or integer multiples of one),
  - if 2, then coefficients are floats in [-1,+1].

**solver.standard_scale_solutions**(nvar, sols, cffs)

Scales the solutions in the list `sols` using the coefficients in `cffs`, using standard double precision arithmetic. The number of variables is given in the parameter `nvar`. If the `sols` are the solutions of the polynomials in the output of `standard_scale_system(pols)`, then the solutions on return will be solutions of the original polynomials in the list `pols`.

**solver.standard_scale_system**(pols)

Applies equation and variable scaling in standard double precision to the polynomials in the list `pols`. On return is the list of scaled polynomials and the scaling coefficients.

**solver.standard_solve**(pols, verbose=True, tasks=0, myfocus=0, dictionary_output=False, verbose_level=0)

Calls the blackbox solver to compute all isolated solutions in standard double precision. On input in `pols` is a list of strings. By default, the solver will print to screen the computed root counts. To make the solver silent, set the flag `verbose` to False. The number of tasks for multithreading is given by `tasks`. The default zero value for `tasks` indicates no multithreading. If the `myfocus` 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. If `dictionary_output`, then on return is a list of dictionaries, else the returned list is a list of strings. If `verbose_level` is larger than 0, then the names of the procedures called in the running of the blackbox solver will be listed. The solving happens in standard double precision arithmetic.

**solver.standard_usolve**(pol, mxi, eps)

Applies the method of Durand-Kerner (aka Weierstrass) to the polynomial in the string `pol`, in standard double precision. The maximum number of iterations is in `mxi`, the requirement on the accuracy in `eps`. 

96 Chapter 4. Reference Manual
solver.test()
Runs test_polyhedral_homotopy, test_solver and test_deflate.

solver.test_deflate()
Applies the deflation method to a system used as example in the paper by T. Ojika on Modified deflation algorithm for the solution of singular problems. I. A system of nonlinear algebraic equations, which appeared in J. Math. Anal. Appl. 123, 199-221, 1987. The approximate solutions were computed via homotopy continuation. The function solve() deflates automatically.

solver.test_dobldobl_polyhedral_homotopy()
Test polyhedral homotopy in double double precision on a small random polynomial system.

solver.test_mixed_volume()
Runs a test on the mixed volume calculators.

solver.test_multiplicity(precision='d')

solver.test_newton()
Tests Newton’s method on simple polynomial system, refining the square root of 2 with increasing precision.

solver.test_newton_laurent()
Tests Newton’s method on simple Laurent system, refining the square root of 2 with increasing precision.

solver.test_nonlinear_reduce()
Tests nonlinear reduction on a simple example.

solver.test_polyhedral_homotopy(precision='d')
Test polyhedral homotopy on small random systems for standard double precision (d), double double precision (dd), or quad double precision (qd).

solver.test_quaddobl_polyhedral_homotopy()
Test polyhedral homotopy in quad double precision on a small random polynomial system.

solver.test_reduce(precision='d')
Tests the reduction of the coefficient matrix of a system.

solver.test_scale()
Performs a basic test on variable scaling.

solver.test_solver()
Generates a random trinomial system and solves it.

solver.test_usolve()
Does a simple sanity check on solving a univariate polynomial at various levels of precision.

solver.total_degree(pols)
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.

solver.total_degree_start_system(pols, checkin=True)
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.

4.1. a blackbox solver for isolated solutions

97
solver.usolve((pol, mxi, eps, precision='d', decimals=100))

Applies the method of Durand-Kerner (aka Weierstrass) to the polynomial in the string pol. The maximum number of iterations is in mxi, the requirement on the accuracy in eps. Four 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)).

mp: arbitrary precision, where the number of decimal places in the working precision is determined by decimals.

4.1.2 functions in the module solutions

The documentation strings of the functions exported by the module solutions are listed below. The script test() runs when typing python solutions.py at the command prompt.

The module solutions exports functions to convert a list of PHCpack solution strings into Python dictionaries. The module exports the definition of the class Solution, as an object-oriented representation of a solution.

class solutions.Solution(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.condition_tables(sols)
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)
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.

dsolutions.diagnostics(sol)
Extracts the diagnostics (err, rco, res) from the PHCpack string solution in sol and returns a triplet of three floats.

dsolutions.endmultiplicity(sol)
Returns the value of t at the end and the multiplicity as (t,m) for the PHCpack solution string sol.
solutions.evaluate(pols, dsol)
Evaluates a list of polynomials given as string in pols at the solution in dictionary format in dsol.

solutions.evaluate_polynomial(pol, dsol)
Evaluates the polynomial pol at the solution dictionary dsol by string substitution.

solutions.filter_real(sols, tol, oper)
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)
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)
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)
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')
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.is_real(sol, tol)
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)
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.make_solution** *(names, values, err=0.0, rco=1.0, res=0.0, tval=0, multiplicity=1)*

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:

```python
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)*

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.numerals** *(dsol)*

Given the dictionary format of a solution `dsol`, returns the list of numeric values of the variables in the solution.

**solutions.str2complex** *(scn)*

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)*

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)*

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')*

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_class** ()

Tests the methods in the class Solution.

**solutions.test_functions** ()

Generates a random trinomial system, solves it, converts the solutions, and then sums the multiplicities.

**solutions.variables** *(dsol)*

Given the dictionary format of a solution in `dsol`, returns the list of variables.
4.1.3 the module polynomials

The module polynomials exports the definition of a class to represent systems of polynomials. The class Polynomials is intended as the starting point of the object-oriented interface to the solver.

The module polynomials exports the definition of the class Polynomials, as the object oriented interface to PHCpack.

class polynomials.Polynomials(pols)

An instance of this class is a list of polynomials, which represents a polynomial system in several variables, with coefficients viewed by default as complex numbers.

solve(verbosetrue,nbtasks=0,mvfocus=0,dictionary_output=False,verbose_level=0)

Applies the blackbox solver and returns a list of solutions. By default, verbose is True, and root counts are written. Multitasking is applied when the number of tasks in nbtasks is set to the number of tasks. To apply only mixed volumes and polyhedral homotopies, set the value for mvfocus to 1. If dictionary_output, then on return is a list of dictionaries, else the returned list is a list of strings. If verbose_level is larger than 0, then the names of the procedures called in the running of the blackbox solver will be listed. The solving happens in standard double precision arithmetic.

variables()

Returns the list of the variables in the polynomials.

polynomials.test()

Tests the methods in the class Polynomials.

4.2 path trackers and sweep homotopies

In the path tracking, we distinguish between paths defined by

1. homotopies with sufficiently random complex numbers, which then are free of singular solutions, except perhaps at the end; and

2. homotopies with real coefficients and real parameters, which most likely may contain singularities.

Functions to track solution paths defined by complex artificial parameter homotopies are exported by the module trackers while the module sweepers exports path trackers for real natural parameter homotopies. The module tuning helps to manage the tolerances of the function to track the solution paths in the trackers module. Access to polyhedral end games is provided by the tropisms module.

4.2.1 functions in the module trackers

The documentation strings of the functions exported by the module trackers are listed below.

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. The path tracking functions in this module can track all solution paths in several levels of precision: standard double, double double, quad double, or arbitrary multiprecision arithmetic. For standard double, double double, and quad double arithmetic, multitasking is supported which could give a good speedup if sufficiently many cores are available on the processor. The tuning of the parameters for the predictor, corrector, and the settings of the tolerances is handled by the tuning module.

trackers.ade_double_double_track(target,start,sols,gamma=0,verbose=1)

Does path tracking with algorithmic differentiation, 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 sols is a list of strings representing start solutions. On return are the string
representations of the solutions computed at the end of the paths. If \( \gamma \) on input equals zero, then a random complex number is generated, otherwise the real and imaginary parts of \( \gamma \) are used.

**trackers.ade_double_track** (\( \text{target, start, sols, } \gamma=0, \text{ verbose}=1 \))

Does path tracking with algorithmic differentiation, in standard double precision. On input are a target system, a start system with solutions. The \( \text{target} \) is a list of strings representing the polynomials of the target system (which has to be solved). The \( \text{start} \) is a list of strings representing the polynomials of the start system, with known solutions in \( \text{sols} \). The \( \text{sols} \) is a list of strings representing start solutions. On return are the string representations of the solutions computed at the end of the paths. If \( \gamma \) on input equals zero, then a random complex number is generated, otherwise the real and imaginary parts of \( \gamma \) are used.

**trackers.ade_quad_double_track** (\( \text{target, start, sols, } \gamma=0, \text{ verbose}=1 \))

Does path tracking with algorithmic differentiation, in quad double precision. On input are a target system, a start system with solutions. The \( \text{target} \) is a list of strings representing the polynomials of the target system (which has to be solved). The \( \text{start} \) is a list of strings representing the polynomials of the start system, with known solutions in \( \text{sols} \). The \( \text{sols} \) is a list of strings representing start solutions. On return are the string representations of the solutions computed at the end of the paths. If \( \gamma \) on input equals zero, then a random complex number is generated, otherwise the real and imaginary parts of \( \gamma \) are used.

**trackers.ade_tuned_double_double_track** (\( \text{target, start, sols, pars, } \gamma=0, \text{ verbose}=1 \))

Does path tracking with algorithmic differentiation, in double double precision, with tuned path parameters. On input are a target system, a start system with solutions. The \( \text{target} \) is a list of strings representing the polynomials of the target system (which has to be solved). The \( \text{start} \) is a list of strings representing the polynomials of the start system, with known solutions in \( \text{sols} \). The \( \text{sols} \) is a list of strings representing start solutions. The \( \text{pars} \) is a tuple of 14 values for the path parameters. On return are the string representations of the solutions computed at the end of the paths. If \( \gamma \) on input equals zero, then a random complex number is generated, otherwise the real and imaginary parts of \( \gamma \) are used.

**trackers.ade_tuned_double_track** (\( \text{target, start, sols, pars, } \gamma=0, \text{ verbose}=1 \))

Does path tracking with algorithmic differentiation, in standard double precision, with tuned parameters in \( \text{pars} \). On input are a target system, a start system with solutions. The \( \text{target} \) is a list of strings representing the polynomials of the target system (which has to be solved). The \( \text{start} \) is a list of strings representing the polynomials of the start system, with known solutions in \( \text{sols} \). The \( \text{sols} \) is a list of strings representing start solutions. The \( \text{pars} \) is a tuple of 14 values for the path parameters. On return are the string representations of the solutions computed at the end of the paths. If \( \gamma \) on input equals zero, then a random complex number is generated, otherwise the real and imaginary parts of \( \gamma \) are used.

**trackers.ade_tuned_quad_double_track** (\( \text{target, start, sols, pars, } \gamma=0, \text{ verbose}=1 \))

Does path tracking with algorithmic differentiation, in quad double precision, with tuned path parameters. On input are a target system, a start system with solutions. The \( \text{target} \) is a list of strings representing the polynomials of the target system (which has to be solved). The \( \text{start} \) is a list of strings representing the polynomials of the start system, with known solutions in \( \text{sols} \). The \( \text{sols} \) is a list of strings representing start solutions. The \( \text{pars} \) is a tuple of tuned values for the path parameters. On return are the string representations of the solutions computed at the end of the paths. If \( \gamma \) on input equals zero, then a random complex number is generated, otherwise the real and imaginary parts of \( \gamma \) are used.

**trackers.cyclic3homotopy**()

Returns a triplet with the cyclic 3-roots system as first item, then in second place a random coefficient system, and the start solutions in the third item of the triplet.

**trackers.double_double_crude_track** (\( \text{target, start, sols, } \gamma=0, \text{ verbose}=\text{True} \))

A crude path tracker does not refine or postprocess the solutions at the end of the paths, computed in double double precision. The \( \text{target} \) is a list of strings representing the polynomials of the target system (which has to be solved). The \( \text{start} \) is a list of strings representing the polynomials of the start system, with known solutions in \( \text{sols} \). The \( \text{sols} \) is a list of strings representing start solutions. By default, a random \( \gamma \) constant is generated, otherwise \( \gamma \) must be a nonzero complex constant. If \( \text{verbose} \), then the solution vectors are written to screen. On return are the string representations of the solutions computed at the end of the paths.
trackers.

**trackers.double_double_track** (target, start, sols, gamma=0, pwt=2, tasks=0)

Does path tracking in double double precision. On input are a target system, a start system with solutions, optionally a \((\text{random})\) 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 **sols** 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 are the string representations of the solutions computed at the end of the paths.

**trackers.gpu_double_double_track** (target, start, sols, gamma=0, verbose=1)

GPU accelerated path tracking with algorithmic differentiation, 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 **sols** is a list of strings representing start solutions. On return are the string representations of the solutions computed at the end of the paths. If gamma on input equals zero, then a random complex number is generated, otherwise the real and imaginary parts of gamma are used.

**trackers.gpu_double_track** (target, start, sols, gamma=0, verbose=1)

GPU accelerated path tracking with algorithmic differentiation, in standard 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 **sols** is a list of strings representing start solutions. On return are the string representations of the solutions computed at the end of the paths. If gamma on input equals zero, then a random complex number is generated, otherwise the real and imaginary parts of gamma are used.

**trackers.gpu_quad_double_track** (target, start, sols, gamma=0, verbose=1)

GPU accelerated path tracking with algorithmic differentiation, 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 **sols** is a list of strings representing start solutions. On return are the string representations of the solutions computed at the end of the paths. If gamma on input equals zero, then a random complex number is generated, otherwise the real and imaginary parts of gamma are used.

**trackers.initialize_dobldobl_solution** (nvar, sol)

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 nvar.

**trackers.initialize_dobldobl_tracker** (target, start, fixedgamma=True, regamma=0.0, imgamma=0.0)

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.

**trackersinitialize_multprec_solution** (nvar, sol)

A multiprecision path tracker with a generator is initialized with a start solution sol in a number of variables equal to the value of nvar.

**trackers.initialize_multprec_tracker** (target, start, fixedgamma=True, decimals=100)

Initializes a path tracker with a generator for a target and start system given in arbitrary multiprecision, with the number of decimal places in the working precision given by the value of decimals. If fixedgamma, then gamma will be a fixed default value, otherwise, a random complex constant for gamma is generated.

**trackers.initialize_quaddobl_solution** (nvar, sol)

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 `nvar`.

```python
trackers.initialize_quaddobl_tracker(target, start, fixedgamma=True, regamma=0.0,  
                                       imgamma=0.0)
```

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.

```python
trackers.initialize_standard_solution(nvar, sol)
```

A standard double precision path tracker with a generator is initialized with a start solution `sol` in a number of variables equal to the value of `nvar`.

```python
trackers.initialize_standard_tracker(target, start, fixedgamma=True, regamma=0.0,  
                                      imgamma=0.0)
```

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.

```python
trackers.initialize_varbprec_solution(nvar, sol)
```

A variable precision path tracker with a generator is initialized with a start solution `sol` in a number of variables equal to the value of `nvar`.

```python
trackers.initialize_varbprec_tracker(target, start, fixedgamma=True)
```

Initializes a path tracker in variable precision with a `target` and `start` system, given as lists of string representations of multivariate polynomials. If `fixedgamma`, then gamma will be a fixed default value, otherwise, a random complex constant for gamma is generated.

```python
trackers.multiprecision_track(target, start, sols, gamma=0, pwt=2, decimals=80)
```

Does path tracking with multiprecision. On input are a target system, a start system with solutions, and optionally a (random) gamma constant. 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. 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 decimal places in the working precision is given by the value of `decimals`. On return are the string representations of the solutions computed at the end of the paths.

```python
trackers.next_dobldobl_solution()
```

Returns the next solution on a path tracked with double double precision arithmetic, provided the functions `initialize_dobldobl_tracker()` and `initialize_dobldobl_solution()` have been executed properly.

```python
trackers.next_multprec_solution()
```

Returns the next solution on a path tracked with arbitrary multiprecision arithmetic, provided the functions `initialize_multprec_tracker()` and `initialize_multprec_solution()` have been executed properly.

```python
trackers.next_quaddobl_solution()
```

Returns the next solution on a path tracked with quad double precision arithmetic, provided the functions `initialize_quaddobl_tracker()` and `initialize_quaddobl_solution()` have been executed properly.

```python
trackers.next_standard_solution()
```

Returns the next solution on a path tracked with standard double precision arithmetic, provided the functions `initialize_standard_tracker()` and `initialize_standard_solution()` have been executed properly.

```python
trackers.next_varbprec_solution(wanted, maxprec, maxit, verbose)
```

Returns the next solution on a path tracked with variable precision arithmetic, provided the functions `initialize_varbprec_tracker()` and `initialize_varbprec_solution()` have been executed properly.
ize_varbprec_tracker() and initialize_varbprec_solution() have been executed properly. The four input parameters are

1. **wanted**: the number of correct decimal places in the solution,
2. **maxprec**: upper bound on the number of decimal places in the precision,
3. **maxit**: maximum number of iterations, and
4. **verbose**: flag to indicate if intermediate output is wanted.

**trackers.quad_double_crude_track** (target, start, sols, gamma=0, verbose=True)

A crude path tracker does not refine or postprocess the solutions at the end of the paths, computed in quad double precision. 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. By default, a random gamma constant is generated, otherwise gamma must be a nonzero complex constant. If **verbose**, then the solution vectors are written to screen. On return are the string representations of the solutions computed at the end of the paths.

**trackers.quad_double_track** (target, start, sols, gamma=0, pwt=2, tasks=0)

Does path tracking with 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 **sols** 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 are the string representations of the solutions computed at the end of the paths.

**trackers.standard_double_crude_track** (target, start, sols, gamma=0, verbose=True)

A crude path tracker does not refine or postprocess the solutions at the end of the paths, computed in standard double precision. 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. By default, a random gamma constant is generated, otherwise gamma must be a nonzero complex constant. If **verbose**, then the solution vectors are written to screen. On return are the string representations of the solutions computed at the end of the paths.

**trackers.standard_double_track** (target, start, sols, gamma=0, pwt=2, tasks=0)

Does path tracking with standard 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 **sols** 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 are the string representations of the solutions computed at the end of the paths.

**trackers.test()**

Runs test_track(), test_next_track(), and test_monitored_track().

**trackers.test_ade_double_double_track**()

Tests the path tracker on the cyclic 3-roots problem, in double double precision.

**trackers.test_ade_double_track**()

Tests the path tracker on the cyclic 3-roots problem, in standard double precision.

**trackers.test_ade_quad_double_track**()

Tests the path tracker on the cyclic 3-roots problem, in quad double precision.
trackers.test_crude_tracker (precision='d')
Runs the crude path trackers. Values for precision are ‘d’, ‘dd’, or ‘qd’, respectively for double, double double,
or quad double precision.

trackers.test_monitored_track()
Often the number of paths to track can be huge and waiting on the outcome of track() without knowing how
many paths that have been tracked so far can be annoying. This script illustrates how one can monitor the
progress of the path tracking. We must use the same gamma constant with each call of track.

trackers.test_next_track (precision='d', decimals=80)
Tests the step-by-step tracking of a solution path. 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)). mp : arbitrary multiprecision with as many decimal places in the working precision as
the value set by decimals.

trackers.test_track (verbose=True, precision='d', decimals=80)
Tests the path tracking on a small random system. Two random trinomials are generated and random constants
are added to ensure there are no singular solutions so we can use this generated system as a start system. The
target system has the same monomial structure as the start system, but with random real coefficients. Because
all coefficients are random, the number of paths tracked equals the mixed volume of the system.

trackers.track (target, start, sols, precision='d', decimals=80, gamma=0, pwt=2, tasks=0)
Runs the path trackers to track solutions in sols at the start system in start to the target system in the
target list using the current settings of the numerical continuation parameters as tuned by the function
tune_track_parameters() of the tuning module. Four levels of precision are supported:
  1. d: standard double precision (1.1e-15 or 2^(-53)),
  2. dd: double double precision (4.9e-32 or 2^(-104)),
  3. qd: quad double precision (1.2e-63 or 2^(-209)).
  4. mp: arbitrary multiprecision, with as many decimal places in the working precision as the value of deci-
mals.

The next to last parameter is optional. By default, a random complex number will be used for gamma, otherwise,
gamma can be any nonzero complex number. 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 last parameter equals
the number of tasks. By default, for tasks equal to 0 there is no multitasking. For positive values of tasks, the
multitasking could give a speedup of up to the number of tasks, depending how many cores are available on the
processor.

4.2.2 functions in the module sweepers
The documentation strings of the functions exported by the module sweepers are listed below.

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. The algorithms applied in this module are
described in the paper by Kathy Piret and Jan Verschelde: Sweeping Algebraic Curves for Singular Solutions. Journal

sweepers.complex_sweep_test (precision='d')
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.doblodobl_complex_sweep(pols, sols, nvar, pars, start, target)`

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.doblodobl_real_sweep(pols, sols, par='s', start=0.0, target=1.0)`

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.quaddobl_complex_sweep(pols, sols, nvar, pars, start, target)`

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 quad 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.quaddobl_real_sweep(pols, sols, par='s', start=0.0, target=1.0)`

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.real_sweep_test(precision='d')`

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.standard_complex_sweep(pols, sols, nvar, pars, start, target)`

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 standard 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.standard_real_sweep(pols, sols, par='s', start=0.0, target=1.0)`

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)*(l - L[0]) + s*(l - L[1]) = 0 so that,

at \( s = 0 \), the natural parameter \( l \) has the value \( L[0] \), and

at \( s = 1 \), the natural parameter \( l \) has the value \( L[1] \).

Thus: as \( s \) moves from 0 to 1, \( l \) goes from \( L[0] \) to \( L[1] \).

All solutions in the list \( \textit{sols} \) must have then the value \( L[0] \) for the variable \( l \). The sweep stops when the target value for \( s \) is reached or when a singular solution is encountered. Computations happen in standard double precision.

### 4.2.3 functions in the module tuning

The documentation strings of the functions exported by the module \texttt{tuning} are listed below.

The module \texttt{tuning} provides functions to tune the tolerances and settings of the predictor and corrector parameters for the path trackers.

- \texttt{tuning.condition_level_get()} \hspace{1cm} Returns an integer that represents the difficulty level of the homotopy. The default level equals zero, higher values lead to smaller tolerances.

- \texttt{tuning.condition_level_set(lvl)} \hspace{1cm} Sets the parameter that represents the difficulty level of the homotopy to the value of \( \textit{lvl} \). The default level equals zero, higher values lead to smaller tolerances. On return is the failure code, which is zero if all went well.

- \texttt{tuning.corrector_abscorendgame_get()} \hspace{1cm} Returns the value of the tolerance on the absolute correction for the corrector during the end game.

- \texttt{tuning.corrector_abscoreonpath_get()} \hspace{1cm} Returns the value of the tolerance on the absolute correction for the corrector along a path, before the end game.

- \texttt{tuning.corrector_abscoreonpath_set(tol)} \hspace{1cm} Sets the tolerance on the absolute correction for the corrector along a path, before the start of the end game, to the value of \( \textit{tol} \). On return is the failure code, which equals zero if all went well.

- \texttt{tuning.corrector_absresendgame_get()} \hspace{1cm} Returns the value of the tolerance on the absolute residual for the corrector during the end game.

- \texttt{tuning.corrector_absresendgame_set(tol)} \hspace{1cm} Sets the tolerance on the absolute residual for the corrector during the end game, to the value of \( \textit{tol} \). On return is the failure code, which equals zero if all went well.

- \texttt{tuning.corrector_absresonpath_get()} \hspace{1cm} Returns the value of the tolerance on the absolute residual for the corrector along a path.

- \texttt{tuning.corrector_absresonpath_set(tol)} \hspace{1cm} Sets the tolerance on the absolute residual for the corrector along a path, before the start of the end game, to the value of \( \textit{tol} \). On return is the failure code, which equals zero if all went well.

- \texttt{tuning.corrector_maxiterendgame_get()} \hspace{1cm} Returns the maximum number of iterations the corrector does along a path, during the end game. The default equals 3.

- \texttt{tuning.corrector_maxiterendgame_set(maxiter)} \hspace{1cm} Sets the maximum number of iterations the corrector does along a path, before the start of the end game, to the value of \( \textit{maxiter} \). On return is the failure code, which equals zero if all went well.

- \texttt{tuning.corrector_maxiteronpath_get()} \hspace{1cm} Returns the maximum number of iterations the corrector does along a path, before the start of the end game. The default equals 3.
tuning.**corrector_maxiteronpath_set**(maxiter)

Sets the maximum number of iterations the corrector does along a path, before the start of the end game, to the value of `maxiter`. On return is the failure code, which equals zero if all went well.

tuning.**corrector_relcorendgame_get**()

Returns the value of the tolerance on the relative correction for the corrector during the end game.

tuning.**corrector_relcorendgame_set**(tol)

Sets the tolerance on the relative correction for the corrector during the end game, to the value of `tol`. On return is the failure code, which equals zero if all went well.

tuning.**corrector_relcoronpath_get**()

Returns the value of the tolerance on the relative correction for the corrector along a path, before the end game.

tuning.**corrector_relcoronpath_set**(tol)

Sets the tolerance on the relative correction for the corrector along a path, before the start of the end game, to the value of `tol`. On return is the failure code, which equals zero if all went well.

tuning.**corrector_relresendgame_get**()

Returns the value of the tolerance on the relative residual for the corrector during the end game.

tuning.**corrector_relresendgame_set**(tol)

Sets the tolerance on the relative residual for the corrector during the end game, to the value of `tol`. On return is the failure code, which equals zero if all went well.

tuning.**corrector_relresonpath_get**()

Returns the value of the tolerance on the relative residual for the corrector along a path, before the end game.

tuning.**corrector_relresonpath_set**(tol)

Sets the tolerance on the relative residual for the corrector along a path, before the start of the end game, to the value of `tol`. On return is the failure code, which equals zero if all went well.

4.2. path trackers and sweep homotopies

---

**default_path_parameters**(precision)

Given in precision 16, 32, or 64 for double, double double, or quad double precision respectively, returns a tuple with the default values for the path parameters.

**distance_to_endgame_get**()

Returns the distance to start the end game. During the end game, the path tracker may apply tolerances that are more severe as the solution paths get more interesting near the end. The default value is 0.1.

**distance_to_endgame_set**(dst)

Sets the distance to start the end game to the value of `dst`. During the end game, the path tracker may apply tolerances that are more severe as the solution paths get more interesting near the end. The default value is 0.1. On return is the failure code, which equals zero if all went well.

**max_steps_get**()

Returns the maximum number of steps the path tracker will perform to reach the end of a solution path. For paths that diverge to infinity are often truncated before reaching extreme values.

**max_steps_set**(mxs)

Sets the maximum number of steps the path tracker will perform to reach the end of a solution path to `mxs`. For paths that diverge to infinity are often truncated before reaching extreme values. On return is the failure code, which equals zero if all went well.

**maximum_reruns_get**()

Returns the value of the maximum number of path reruns. If path jumping is detected, then the clustered paths are retracked with more severe values of the tolerances. The default value equals one.

**maximum_reruns_set**(mrr)

Sets the value of the maximum number of path reruns to the value of `mrr`. If path jumping is detected, then the
clustered paths are retracked with more severe values of the tolerances. The default value equals one. On return is the failure code, which is zero when all went well.

**tuning.** `order_endgame_extrapolator_get()`
Returns the order of the extrapolator to estimate the winding number in a polyhedral end game. If the order is zero, then no polyhedral end game will be applied.

**tuning.** `order_endgame_extrapolator_set(ord)`
Sets the order of the extrapolator to estimate the winding number in a polyhedral end game to the value of `ord`. If the order `ord` is zero, then no polyhedral end game will be applied. On return is the failure code, which equals zero if all went well.

**tuning.** `predictor_expfacendgame_get()`
Returns the value of the expansion factor to increase the step size in case of a successful corrector stage, during the end game. The expansion factor determines the speed at which the predictor increases its step size when tracking an easier portion of the path.

**tuning.** `predictor_expfacendgame_set(expfac)`
Sets the value of the expansion factor to increase the step size in case of a successful corrector stage, during the end game, to the value of `expfac`. On return is the failure code, which equals zero if all went well.

**tuning.** `predictor_expfaconpath_get()`
Returns the value of the expansion factor to increase the step size in case of a successful corrector stage, along a path, before the end game. The expansion factor determines the speed at which the predictor increases its step size when tracking an easier portion of the path.

**tuning.** `predictor_expfaconpath_set(expfac)`
Sets the value of the expansion factor to increase the step size in case of a successful corrector stage, along a path, before the end game, to the value of `expfac`. On return is the failure code, which equals zero if all went well.

**tuning.** `predictor_exptrsendgame_get()`
Returns the value of the expansion threshold for the step size control, during the end game. The expansion threshold is the number of consecutive successful corrector stages that must be met before the step size is increased.

**tuning.** `predictor_exptrsendgame_set(exprs)`
Sets the value of the expansion threshold for the step size control, during the end game, to the value of `exprs`. The expansion threshold is the number of consecutive successful corrector stages that must be met before the step size is increased. On return is the failure code, which equals zero if all went well.

**tuning.** `predictor_exptrsonpath_get()`
Returns the value of the expansion threshold for the step size control, along a path, before the end game. The expansion threshold is the number of consecutive successful corrector stages that must be met before the step size is increased.

**tuning.** `predictor_exptrsonpath_set(exprs)`
Sets the value of the expansion threshold for the step size control, along a path, before the end game, to the value of `exprs`. The expansion threshold is the number of consecutive successful corrector stages that must be met before the step size is increased. On return is the failure code, which equals zero if all went well.

**tuning.** `predictor_maxstependgame_get()`
Returns the maximum value of the step size during the end game. The step size control will never increase the step size to a value above this maximum.

**tuning.** `predictor_maxsteponpath_get()`
Returns the maximum value of the step size along a path, before the end game. The step size control will never increase the step size to a value above this maximum.

**tuning.** `predictor_maxsteponpath_set(maxstep)`
Sets the maximum of the step size along a path before the end game to the value of `maxstep`. On return is the
failure code, which equals zero if all went well.

tuning.predictor_maxtependgame_set(maxstep)
Sets the maximum of the step size during the end game to the value of maxstep. On return is the failure code, which equals zero if all went well.

tuning.predictor_minsteependgame_get()
Returns the minimum value of the step size during the end game. If the step size control cuts the step size to a value below this minimum, then the path tracking is aborted.

tuning.predictor_minsteependgame_set(minstep)
Sets the minimum of the step size during the end game to the value of minstep. On return is the failure code, which equals zero if all went well.

tuning.predictor_minsteptonpath_get()
Returns the minimum value of the step size along a path, before the end game. If the step size control cuts the step size to a value below this minimum, then the path tracking is aborted.

tuning.predictor_minsteptonpath_set(minstep)
Sets the minimum of the step size along a path to the value of minstep. On return is the failure code, which equals zero if all went well.

tuning.predictor_redfacendgame_get()
Returns the value of the reduction factor to cut the step size in case of a failed corrector stage, during the end game. The reduction factor determines the speed at which the predictor reduces its step size when tracking a more difficult portion of the path.

tuning.predictor_redfacendgame_set(redfac)
Sets the value of the reduction factor to cut the step size in case of a failed corrector step during the end game, to the value of redfac. On return is the failure code, which equals zero if all went well.

tuning.predictor_redfaconpath_get()
Returns the value of the reduction factor to cut the step size in case of a failed corrector stage, along a path, before the end game. The reduction factor determines the speed at which the predictor reduces its step size when tracking a more difficult portion of the path.

tuning.predictor_redfaconpath_set(redfac)
Sets the value of the reduction factor to cut the step size in case of a failed corrector step, along a path, before the end game, to the value of redfac. On return is the failure code, which equals zero if all went well.

tuning.predictor_typeendgame_get()
Returns an integer which represents the type of the predictor along a path, before the start of the end game. The integer on return takes values between 0 and 9, depending on the type for the solution x and for the continuation parameter t. The ten predictor types are

0 : secant for x, real for t;
1 : secant for x, complex for t;
2 : secant for x, geometric for t;
3 : tangent for x, real for t;
4 : tangent for x, complex for t;
5 : tangent for x, geometric for t;
6 : Hermite for x, real for t;
7 : quadratic for x, real for t;
8 : cubic for x, real for t;
9 : quartic for x, real for t.
tuning.\texttt{predictor_typeendgame_set (predtype)}

Sets the type of the predictor during the end game to what the value of \texttt{predtype} represents. A valid integer value for \texttt{predtype} lies between 0 and 9, setting the type for the solution x and for the continuation parameter t. The ten predictor types are

0 : secant for x, real for t;
1 : secant for x, complex for t;
2 : secant for x, geometric for t;
3 : tangent for x, real for t;
4 : tangent for x, complex for t;
5 : tangent for x, geometric for t;
6 : Hermite for x, real for t;
7 : quadratic for x, real for t;
8 : cubic for x, real for t;
9 : quartic for x, real for t.

On return is the failure code, which is zero when all went well.

\textbf{tuning.\texttt{predictor_typeonpath_get ()}}

Returns an integer which represents the type of the predictor along a path, before the start of the end game. The integer on return takes values between 0 and 9, depending on the type for the solution x and for the continuation parameter t. The ten predictor types are

0 : secant for x, real for t;
1 : secant for x, complex for t;
2 : secant for x, geometric for t;
3 : tangent for x, real for t;
4 : tangent for x, complex for t;
5 : tangent for x, geometric for t;
6 : Hermite for x, real for t;
7 : quadratic for x, real for t;
8 : cubic for x, real for t;
9 : quartic for x, real for t.

\textbf{tuning.\texttt{predictor_typeonpath_set (predtype)}}

Sets the type of the predictor along a path, before the end game, to what the value of \texttt{predtype} represents. A valid integer value for \texttt{predtype} lies between 0 and 9, setting the type for the solution x and for the continuation parameter t. The ten predictor types are

0 : secant for x, real for t;
1 : secant for x, complex for t;
2 : secant for x, geometric for t;
3 : tangent for x, real for t;
4 : tangent for x, complex for t;
5 : tangent for x, geometric for t;
phcpy Documentation, Release 1.1.1

6: Hermite for x, real for t;
7: quadratic for x, real for t;
8: cubic for x, real for t;
9: quartic for x, real for t.

On return is the failure code, which is zero when all went well.

tuning.set_path_parameter_value(idx)
Given the index idx of a path parameter, prompts the user for a new value which will be returned.

tuning.test()
Tests the tuning of the parameters.

tuning.tolerance_clustsolendgame_get()
Returns the tolerance on the distance between two solutions to decide whether two solutions are clustered during the end game.

tuning.tolerance_clustsolendgame_set(tol)
Sets the tolerance on the distance between two solutions to decide whether two solutions are clustered during the end game, to the value of tol. On return is the failure code which equals zero if all went well.

tuning.tolerance_clustsolonpath_get()
Returns the tolerance on the distance between two solutions to decide whether two solutions are clustered along a path, before the end game.

tuning.tolerance_clustsolonpath_set(tol)
Sets the tolerance on the distance between two solutions to decide whether two solutions are clustered along a path, before the end game, to the value of tol. On return is the failure code which equals zero if all went well.

tuning.tolerance_infsolendgame_get()
Returns the tolerance threshold to decide whether a solution path diverges to infinity, during the end game.

tuning.tolerance_infsolendgame_set(tol)
Sets the tolerance threshold to decide whether a solution path diverges to infinity, during the end game, to the value of tol. On return is the failure code which is zero if all went well.

tuning.tolerance_infsolonpath_get()
Returns the tolerance threshold to decide whether a solution path diverges to infinity, before the start of the end game.

tuning.tolerance_infsolonpath_set(tol)
Sets the tolerance threshold to decide whether a solution path diverges to infinity, before the end game, to the value of tol. On return is the failure code which is zero if all went well.

tuning.tolerance_rcondendgame_get()
Returns the tolerance on the inverse condition number of the Jacobian matrix of a solution, during the end game, to decide whether a solution is singular or not.

tuning.tolerance_rcondendgame_set(tol)
Sets the tolerance on the inverse condition number of the Jacobian matrix of a solution along a path, before the end game, to decide whether a solution is singular, to the value of tol. On return is the failure code, which equals zero if all went well.

tuning.tolerance_rcondonpath_get()
Returns the tolerance on the inverse condition number of the Jacobian matrix of a solution along a path, before the end game, to decide whether a solution is singular.

tuning.tolerance_rcondonpath_set(tol)
Sets the tolerance on the inverse condition number of the Jacobian matrix of a solution along a path, before the end game.
end game, to decide whether a solution is singular, to the value of tol. The failure code is returned, which is zero if all went well.

**tuning.track_simultaneously_get()**

Returns the number of paths that are tracked simultaneously, for the same discretization of the interval of the continuation parameter. The default value equals one. Increasing this value avoids path crossing, also called path jumping. This jumping happens when the approximated points on one path transition to approximated points on another path.

**tuning.track_simultaneously_set(nbr)**

Sets the number of paths that are tracked simultaneously, for the same discretization of the interval of the continuation parameter, to the value of nbr. The default value equals one. Increasing this value avoids path crossing, also called path jumping. This jumping happens when the approximated points on one path transition to approximated points on another path. On return in the failure code, which is zero if all went well.

**tuning.tune_path_parameters(precision)**

Given in precision the value 16, 32, or 64 for double, double double, or quad double precision respectively, tunes the path parameters interactively, starting from the default path parameters.

**tuning.tune_track_parameters(difficulty=0, digits=16, interactive=False, silent=True)**

Tunes the numerical continuation parameters for use during path tracking based on two parameters: the difficulty of the solution path (difficulty) and the number of decimal places (digits) for the accuracy of the approximations along a path. Increasing difficulty will decrease the step size and increase the number of steps along a path. Increasing digits will decrease the tolerances on the numerical approximations. If interactive is True, then the user can interactively adjust specific numerical continuation parameters. If silent is False, then the current values of the numerical continuation parameters are shown.

**tuning.write_path_parameters(pars)**

Given in pars is a 14-tuple with the path parameters. The path parameters are written, for later interactive tuning.

### 4.2.4 functions in the module tropisms

The module tropisms provides access to the numerically computed tropisms via a polyhedral end game. The functions exported by the module tropisms are listed below.

The module tropisms exports function to manage numerically computed tropisms in double, double double, or quad double precision.

**tropisms.doblodobl_clear()**

Clears the tropisms stored in double double precision.

**tropisms.doblodobl_dimension()**

Returns the dimension of the tropisms stored in double double precision.

**tropisms.doblodobl_initialize(nbt, dim, wnd, dir, err)**

Initializes the direction vectors computed 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;
- **dir**: a list of lists of double doubles with the coordinates of the directions, each inner list has dim double doubles and nbt vectors;
- **err**: a list of nbt double doubles.
tropisms.\texttt{dobldobl\_retrieve}(nbt, dim)
Given on input the number of tropisms in \textit{nbt} and the dimension in \textit{dim}, returns a tuple of three lists: the winding numbers, coordinates of the direction vectors, and the errors; in double double precision.

tropisms.\texttt{dobldobl\_size}()
Returns the number of tropisms stored in double double precision.

tropisms.\texttt{management\_test}()
Tests the management of numerically computed tropisms.

tropisms.\texttt{quaddobl\_clear}()
Clears the tropisms stored in quad double precision.

tropisms.\texttt{quaddobl\_dimension}()
Returns the dimension of the tropisms stored in quad double precision.

tropisms.\texttt{quaddobl\_initialize}(nbt, dim, wnd, dir, err)
Initializes the direction vectors computed in quad double precision, along with estimates for their winding numbers and errors. On entry are the following five parameters:
\begin{itemize}
\item \textit{nbt}: the number of direction vectors;
\item \textit{dim}: the number of coordinates in each vector;
\item \textit{wnd}: a list of integer values for the winding numbers, as many as \textit{nbt};
\item \textit{dir}: a list of lists of quad doubles with the coordinates of the directions, each inner list has \textit{dim} quad doubles and \textit{nbt} vectors;
\item \textit{err}: a list of \textit{nbt} double doubles.
\end{itemize}

tropisms.\texttt{quaddobl\_retrieve}(nbt, dim)
Given on input the number of tropisms in \textit{nbt} and the dimension in \textit{dim}, returns a tuple of three lists: the winding numbers, coordinates of the direction vectors, and the errors; in quad double precision.

tropisms.\texttt{quaddobl\_size}()
Returns the number of tropisms stored in quad double precision.

tropisms.\texttt{retrieve\_dobldobl\_tropism}(dim, idx)
Returns the winding number, coordinates of the direction, and its error, stored in double double precision, of dimension \textit{dim}, and index \textit{idx}. The index must be in the range 1..\texttt{dobldobl\_size}(). Observe that the index counter starts at one and not at zero.

tropisms.\texttt{retrieve\_quaddobl\_tropism}(dim, idx)
Returns the winding number, coordinates of the direction, and its error, stored in quad double precision, of dimension \textit{dim}, and index \textit{idx}. The index must be in the range 1..\texttt{quaddobl\_size}(). Observe that the index counter starts at one and not at zero.

tropisms.\texttt{retrieve\_standard\_tropism}(dim, idx)
Returns the winding number, coordinates of the direction, and its error, stored in double precision, of dimension \textit{dim}, and index \textit{idx}. The index must be in the range 1..\texttt{standard\_size}(). Observe that the index counter starts at one and not at zero.

tropisms.\texttt{standard\_clear}()
Clears the tropisms stored in standard double precision.

tropisms.\texttt{standard\_dimension}()
Returns the dimension of the tropisms stored in standard double precision.

tropisms.\texttt{standard\_initialize}(nbt, dim, wnd, dir, err)
Initializes the direction vectors computed in double precision, along with estimates for their winding numbers and errors. On entry are the following five parameters:

### 4.2. path trackers and sweep homotopies
\( nbt \): 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 \( nbt \);
\( dir \): a list of lists of doubles with the coordinates of the directions, each inner list has \( dim \) doubles and \( nbt \) vectors are given;
\( err \): a list of \( nbt \) doubles.

\texttt{tropisms.standard_retrieve(nbt, dim)}

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 standard double precision.

\texttt{tropisms.standard_size()}

Returns the number of tropisms stored in standard double precision.

\texttt{tropisms.store_dobldobl_tropism(dim, idx, wnd, dir, err)}

Stores the tropism, given in double double precision, with dim doubles as coordinates in the list \( dir \), the error in \( err \), and the winding number \( wnd \), at position \( idx \). The index \( idx \) must be in the range 1..dobldobl_size().

\texttt{tropisms.store_quaddobl_tropism(dim, idx, wnd, dir, err)}

Stores the tropism, given in quad double precision, with dim doubles as coordinates in the list \( dir \), the error in \( err \), and the winding number \( wnd \), at position \( idx \). The index \( idx \) must be in the range 1..quaddobl_size().

\texttt{tropisms.store_standard_tropism(dim, idx, wnd, dir, err)}

Stores the tropism, given in standard double precision, with dim doubles as coordinates in the list \( dir \), the error in \( err \), and the winding number \( wnd \), at position \( idx \). The index \( idx \) must be in the range 1..standard_size().

\texttt{tropisms.test(precision='d')}  
Tests the numerical computation of a tropism.

\texttt{tropisms.test_dobldobl_store_load()}  
Tests the storing and loading of numerically computed tropisms, in double double precision.

\texttt{tropisms.test_quaddobl_store_load()}  
Tests the storing and loading of numerically computed tropisms, in quad double precision.

\texttt{tropisms.test_standard_store_load()}  
Tests the storing and loading of numerically computed tropisms, in standard double precision.

\section*{4.3 positive dimensional solution sets}

Numerical representations of positive dimensional solution sets are called witness sets and are computed by the functions exported by the module \texttt{sets}. Cascades of homotopies compute generic points on each component of all dimensions. In a numerical irreducible decomposition, all equidimensional solution sets are factored into irreducible components.
4.3.1 functions in the module sets

This module exports routines of PHCpack to manipulate positive dimensional solution sets of polynomial systems. 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.

```
def dobdobl_embed(nvar, topdim, pols)
    Given in pols a list of strings that represent polynomials in nvar 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.
```

```
def dobdobl_ismember(wsys, gpts, dim, point, evatol=1e-06, memtol=1e-06, verbose=True, tasks=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 string point, which is the string representation of a solution in PHCpack format, with symbols of the variables before the values of the coordinates. 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. Returns a tuple of two booleans. The first boolean is True if the point satisfies the equations, otherwise it is False. The second boolean is True if the point belongs to the witness set, otherwise, the second boolean is False.
```

```
def dobdobl_ismember_filter(wsys, gpts, dim, points, rcotol=1e-06, evatol=1e-06, memtol=1e-06, verbose=True, tasks=0)
    Given in wsys and gpts is a witness set of dimension dim, where wsys is an embedded polynomial system, and in points a list of strings. The strings represent points as solutions in PHCpack format. The homotopy membership test is applied to each point in the list points. The list on return contains the points that do NOT belong to the witness set. Points that belong to the witness set are considered junk. 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 parameter rcotol is used to bypass the homotopy membership test, for points with their estimated inverse condition number larger than rcotol will be considered isolated and not in the witness set.
```

```
def dobdobl_laurent_embed(nvar, topdim, pols)
    Given in pols a list of strings that represent Laurent polynomials in nvar 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.
```

```
def dobdobl_laurent_ismember(wsys, gpts, dim, point, evatol=1e-06, memtol=1e-06, verbose=True, tasks=0)
    Applies the homotopy membership test for a point to belong to a witness set of dimension dim, given by an embedding polynomial Laurent system in wsys, with corresponding generic points in gpts. The coordinates of the test point are given in the string point, which is the string representation of a solution in PHCpack format, with symbols of the variables before the values of the coordinates. 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. Returns a tuple of two booleans. The first boolean is True if the point satisfies the equations, otherwise it is False. The second boolean is True if the point belongs to the witness set, otherwise, the second boolean is False.
```

```
def dobdobl_laurent_ismember_filter(wsys, gpts, dim, points, rcotol=1e-06, evatol=1e-06, memtol=1e-06, verbose=True, tasks=0)
    Given in wsys and gpts is a witness set of dimension dim, where wsys is an embedded Laurent polynomial system, and in points a list of strings. The strings represent points as solutions in PHCpack format. The
homotopy membership test is applied to each point in the list points. The list on return contains the points that do NOT belong to the witness set. Points that belong to the witness set are considered junk. 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 parameter rctol is used to bypass the homotopy membership test, for points with their estimated inverse condition number larger than rctol will be considered isolated and not in the witness set.

```
sets.dobldobl_laurent_membertest (wsys, gpts, dim, point, evatol=1e-06, memtol=1e-06, verbose=True, tasks=0)
```

Applies the homotopy membership test for a point to belong to a witness set of dimension dim, given by an embedding 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.dobldobl_membertest (wsys, gpts, dim, point, evatol=1e-06, memtol=1e-06, verbose=True, tasks=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.drop_coordinate_from_dobldobl_solutions (sols, nbvar, svar)
```

Removes the variable with symbol in the string svar from the list sols of strings that represent solutions in nbvar variables, in double double precision.

```
sets.drop_coordinate_from_quaddobl_solutions (sols, nbvar, svar)
```

Removes the variable with symbol in the string svar from the list sols of strings that represent solutions in nbvar variables, in quad double precision.

```
sets.drop_coordinate_from_standard_solutions (sols, nbvar, svar)
```

Removes the variable with symbol in the string svar from the list sols of strings that represent solutions in nbvar variables, in standard double precision.

```
sets.drop_variable_from_dobldobl_laurent_polynomials (pols, svar)
```

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. Note that the system in pols must be square.

```
sets.drop_variable_from_dobldobl_polynomials (pols, svar)
```

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. Note that the system in pols must be square.

```
sets.drop_variable_from_quaddobl_laurent_polynomials (pols, svar)
```

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. Note that the system in pols must be square.

```
sets.drop_variable_from_quaddobl_polynomials (pols, svar)
```

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. Note that the system in pols must be square.

```
sets.drop_variable_from_standard_laurent_polynomials (pols, svar)
```

Removes the variable with symbol in the string svar from the list pols of strings that represent Laurent poly-
mials in several variables, with coefficients in standard double precision. Note that the system in \textit{pols} must be square.

\texttt{sets.drop_variable_from_standard_polynomials(pols, svar)}

Removes the variable with symbol in the string \textit{svar} from the list \textit{pols} of strings that represent polynomials in several variables, with coefficients in standard double precision. Note that the system in \textit{pols} must be square.

\texttt{sets.embed(nvar, topdim, pols, precision='d')}  

Given in \textit{pols} a list of strings that represent polynomials in \textit{nvar} variables, this function returns an embedding of \textit{pols} of dimension \textit{topdim}. The \textit{topdim} is the top dimension which equals the expected highest dimension of a component of the solution set of the system of polynomials. The default \textit{precision} of the coefficients is ‘d’, for standard double precision. For double double and quad double precision, set the value of \textit{precision} to ‘dd’ or ‘qd’ respectively.

\texttt{sets.is_member(wsys, gpts, dim, solpt, evatol=1e-06, memtol=1e-06, verbose=True, precision='d', tasks=0)}

This function wraps the membertest where the point is a solution, given in \textit{solpt}. All other parameters have the same meaning as in the function membertest.

\texttt{sets.is_signed(pol)}

Given in \textit{pol} is the string representation of a polynomial. Returns True if the first nonspace character in the string \textit{pol} is either ‘+’ or ‘-’. Returns False otherwise.

\texttt{sets.is_slackvar(var)}

Given in \textit{var} is a string with a variable name. Returns True if the variable name starts with ‘zz’, followed by a number. Returns False otherwise.

\texttt{sets.ismember_filter(wsys, gpts, dim, points, rcotol=1e-06, evatol=1e-06, memtol=1e-06, verbose=True, precision='d', tasks=0)}

Filters \textit{points} so the list on return contains only those points which do not belong to the witness set of dimension \textit{dim}, given by an embedded polynomial system in \textit{wsys}, with corresponding generic points in \textit{gpts}. The list \textit{points} is a list of strings. Each string is the symbolic string representation of a solution. By default, \textit{verbose} is True, and the precision is double ‘d’. Other levels of precision are double double precision ‘dd’ and quad double precision ‘qd’. The number of threads is given in \textit{tasks}. If \textit{tasks} is zero, then no multithreading is applied in the homotopy membership test. The parameter \textit{rcotol} is used to bypass the homotopy membership test, for points with their estimated inverse condition number larger than \textit{rcotol} will be considered isolated and not in the witness set. The homotopy membership test has two tolerances: \textit{evatol} and \textit{memtol}. The \textit{evatol} is the tolerance on the residual of the evaluation of the polynomial equations at the test point. If the residual of the evaluation is not less than \textit{evatol}, then the point is not a member. Otherwise, the homotopy membership test is called and the \textit{memtol} is used to compare the coordinates of the point with the newly computed generic points. If there is a match between the coordinates within the given tolerance \textit{memtol}, then the points is a member and filtered out.

\texttt{sets.laurent_embed(nvar, topdim, pols, precision='d')}  

Given in \textit{pols} a list of strings that represent Laurent polynomials in \textit{nvar} variables, this function returns an embedding of \textit{pols} of dimension \textit{topdim}. The \textit{topdim} is the top dimension which equals the expected highest dimension of a component of the solution set of the system of polynomials. The default \textit{precision} of the coefficients is ‘d’, for standard double precision. For double double and quad double precision, set the value of \textit{precision} to ‘dd’ or ‘qd’ respectively.

\texttt{sets.laurent_ismember_filter(wsys, gpts, dim, points, rcotol=1e-06, evatol=1e-06, memtol=1e-06, verbose=True, precision='d', tasks=0)}

Filters \textit{points} so the list on return contains only those points which do not belong to the witness set of dimension \textit{dim}, given by an embedded Laurent polynomial system in \textit{wsys}, with corresponding generic points in \textit{gpts}. The list \textit{points} is a list of strings. Each string is the symbolic string representation of a solution. By default, \textit{verbose} is True, and the precision is double ‘d’. Other levels of precision are double double precision ‘dd’ and quad double precision ‘qd’. The number of threads is given in \textit{tasks}. If \textit{tasks} is zero, then no multithreading is applied in the homotopy membership test. The parameter \textit{rcotol} is used to bypass the homotopy membership test, for points with their estimated inverse condition number larger than \textit{rcotol} will be considered isolated and not in the
witness set. The homotopy membership test has two tolerances: \textit{evatol} and \textit{memtol}. The \textit{evatol} is the tolerance on the residual of the evaluation of the polynomial equations at the test point. If the residual of the evaluation is not less than \textit{evatol}, then the point is not a member. Otherwise, the homotopy membership test is called and the \textit{memtol} is used to compare the coordinates of the point with the newly computed generic points. If there is a match between the coordinates within the given tolerance \textit{memtol}, then the point is a member and filtered out.

\texttt{sets\.laurent\_membertest}(\texttt{wsys, gpts, dim, point, evatol=1e-06, memtol=1e-06, verbose=True, precision='d', tasks=0})

Applies the homotopy membership test for a point to belong to a witness set of dimension \textit{dim}, given by an embedding Laurent system in \texttt{wsys}, with corresponding generic points in \texttt{gpts}. The coordinates of the test point are given in the list \texttt{point}, as a list of doubles, with the real and imaginary part of each coordinate of the point. By default, \texttt{verbose} is True. The number of threads is given in \texttt{tasks}. If \texttt{tasks} is zero, then no multithreading is applied in the homotopy membership test. The default working \texttt{precision} is double \textit{\textquote{d}}. Other levels of precision are double double precision \textit{\textquote{dd}} and quad double precision \textit{\textquote{qd}}. There are two tolerances: \textit{evatol} is the tolerance on the residual of the evaluation of the Laurent system at the test point. If the residual of the evaluation is not less than \textit{evatol}, then the \texttt{membertest} returns False. Otherwise, the homotopy membership test is called and the \textit{memtol} is used to compare the coordinates of the point with the newly computed generic points. If there is a match between the coordinates within the given tolerance \textit{memtol}, then True is returned.

\texttt{sets\.membertest}(\texttt{wsys, gpts, dim, point, evatol=1e-06, memtol=1e-06, verbose=True, precision='d', tasks=0})

Applies the homotopy membership test for a point to belong to a witness set of dimension \textit{dim}, given by an embedding polynomial system in \texttt{wsys}, with corresponding generic points in \texttt{gpts}. The coordinates of the test point are given in the list \texttt{point}, as a list of doubles, with the real and imaginary part of each coordinate of the point. By default, \texttt{verbose} is True. The number of threads is given in \texttt{tasks}. If \texttt{tasks} is zero, then no multithreading is applied in the homotopy membership test. The default working \texttt{precision} is double \textit{\textquote{d}}. Other levels of precision are double double precision \textit{\textquote{dd}} and quad double precision \textit{\textquote{qd}}. There are two tolerances: \textit{evatol} is the tolerance on the residual of the evaluation of the polynomial equations at the test point. If the residual of the evaluation is not less than \textit{evatol}, then the \texttt{membertest} returns False. Otherwise, the homotopy membership test is called and the \textit{memtol} is used to compare the coordinates of the point with the newly computed generic points. If there is a match between the coordinates within the given tolerance \textit{memtol}, then True is returned.

\texttt{sets\.quaddobl\_embed}(\texttt{nvar, topdim, polys})

Given in \texttt{polys} a list of strings that represent polynomials in \texttt{nvar} variables, with coefficients in quad double precision, this function returns an embedding of \texttt{polys} of dimension \textit{topdim}. The \textit{topdim} is the top dimension which equals the expected highest dimension of a component of the solution set of the system of polynomials.

\texttt{sets\.quaddobl\_ismember}(\texttt{wsys, gpts, dim, point, evatol=1e-06, memtol=1e-06, verbose=True, precision='d', tasks=0})

Applies the homotopy membership test for a point to belong to a witness set of dimension \textit{dim}, given by an embedding polynomial system in \texttt{wsys}, with corresponding generic points in \texttt{gpts}. The coordinates of the test point are given in the string \texttt{point}, which is the string representation of a solution in PHCpack format, with symbols of the variables before the values of the coordinates. By default, \texttt{verbose} is True. The number of threads is given in \texttt{tasks}. If \texttt{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 \textit{(evatol)} and the membership \textit{(memtol)} allow for singular values at the end points of the paths in the homotopy membership test. Returns a tuple of two booleans. The first boolean is True if the point satisfies the equations, otherwise it is False. The second boolean is True if the point belongs to the witness set, otherwise, the second boolean is False.

\texttt{sets\.quaddobl\_ismember\_filter}(\texttt{wsys, gpts, dim, points, rcotol=1e-06, evatol=1e-06, memtol=1e-06, verbose=True, tasks=0})

Given in \texttt{wsys} and \texttt{gpts} is a witness set of dimension \textit{dim}, where \texttt{wsys} is an embedded polynomial system, and in \texttt{points} a list of strings. The strings represent points as solutions in PHCpack format. The homotopy membership test is applied to each point in the list \texttt{points}. The list on return contains the points that do NOT belong to the witness set. Points that belong to the witness set are considered junk. By default, \texttt{verbose} is True. The number of threads is given in \texttt{tasks}. If \texttt{tasks} is zero, then no multithreading is applied in the homotopy membership test.
Calculations happen in quad double precision. The parameter rcotol is used to bypass the homotopy membership test, for points with their estimated inverse condition number larger than rcotol will be considered isolated and not in the witness set.

```
sets.quaddobl_laurent_embed(nvar, topdim, pols)
```

Given in pols a list of strings that represent Laurent polynomials in nvar 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.

```
sets.quaddobl_laurent_ismember(wsys, gpts, dim, point, evatol=1e-06, memtol=1e-06, verbose=True, tasks=0)
```

Applies the homotopy membership test for a point to belong to a witness set of dimension dim, given by an embedding polynomial Laurent system in wsys, with corresponding generic points in gpts. The coordinates of the test point are given in the string point, which is the string representation of a solution in PHCpack format, with symbols of the variables before the values of the coordinates. 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. Returns a tuple of two booleans. The first boolean is True if the point satisfies the equations, otherwise it is False. The second boolean is True if the point belongs to the witness set, otherwise, the second boolean is False.

```
sets.quaddobl_laurent_ismember_filter(wsys, gpts, dim, points, rcotol=1e-06, evatol=1e-06, memtol=1e-06, verbose=True, tasks=0)
```

Given in wsys and gpts a witness set of dimension dim, where wsys is an embedded Laurent polynomial system, and in points a list of strings. The strings represent points as solutions in PHCpack format. The homotopy membership test is applied to each point in the list points. The list on return contains the points that do NOT belong to the witness set. Points that belong to the witness set are considered junk. 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 parameter rcotol is used to bypass the homotopy membership test, for points with their estimated inverse condition number larger than rcotol will be considered isolated and not in the witness set.

```
sets.quaddobl_laurent_membertest(wsys, gpts, dim, point, evatol=1e-06, memtol=1e-06, verbose=True, tasks=0)
```

Applies the homotopy membership test for a point to belong to a witness set of dimension dim, given by an embedding polynomial 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.quaddobl_membertest(wsys, gpts, dim, point, evatol=1e-06, memtol=1e-06, verbose=True, tasks=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.standard_embed(nvar, topdim, pols)
```

Given in pols a list of strings representing polynomials in nvar variables, with coefficients in standard 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.

```
sets.standard_ismember(wsys, gpts, dim, point, evatol=1e-06, memtol=1e-06, verbose=True, tasks=0)
```

**4.3. positive dimensional solution sets**
Applies the homotopy membership test for a point to belong to a witness set of dimension \( \text{dim} \), given by an embedding polynomial system in \( \text{wsys} \), with corresponding generic points in \( \text{gpts} \). The coordinates of the test point are given in the string \( \text{point} \), which is the string representation of a solution in PHCpack format, with symbols of the variables before the values of the coordinates. By default, \( \text{verbose} \) is True. The number of threads is given in \( \text{tasks} \). If \( \text{tasks} \) is zero, then no multithreading is applied in the homotopy membership test. Calculations happen in standard double precision. The default values for the evaluation (\( \text{evatol} \)) and the membership (\( \text{memtol} \)) allow for singular values at the end points of the paths in the homotopy membership test. Returns a tuple of two booleans. The first boolean is True if the point satisfies the equations, otherwise it is False. The second boolean is True if the point belongs to the witness set, otherwise, the second boolean is False.

\[ \text{sets.standard_ismember_filter}(\text{wsys}, \text{gpts}, \text{dim}, \text{points}, \text{rcotol}=1e-06, \text{evatol}=1e-06, \text{memtol}=1e-06, \text{bose}=\text{True}, \text{tasks}=0) \]

Given in \( \text{wsys} \) and \( \text{gpts} \) is a witness set of dimension \( \text{dim} \), where \( \text{wsys} \) is an embedded polynomial system, and in \( \text{points} \) a list of strings. The strings represent points as solutions in PHCpack format. The homotopy membership test is applied to each point in the list \( \text{points} \). The list on return contains the points that do NOT belong to the witness set. Points that belong to the witness set are considered junk. By default, \( \text{verbose} \) is True. The number of threads is given in \( \text{tasks} \). If \( \text{tasks} \) is zero, then no multithreading is applied in the homotopy membership test. Calculations happen in standard double precision. The parameter \( \text{rcotol} \) is used to bypass the homotopy membership test, for points with their estimated inverse condition number larger than \( \text{rcotol} \) will be considered isolated and not in the witness set.

\[ \text{sets.standard_laurent_embed}(\text{nvar}, \text{topdim}, \text{pols}) \]

Given in \( \text{pols} \) a list of strings representing Laurent polynomials in \( \text{nvar} \) variables, with coefficients in standard double precision, this function returns an embedding of \( \text{pols} \) of dimension \( \text{topdim} \). The \( \text{topdim} \) is the top dimension which equals the expected highest dimension of a component of the solution set of the system of polynomials.

\[ \text{sets.standard_laurent_ismember}(\text{wsys}, \text{gpts}, \text{dim}, \text{point}, \text{evatol}=1e-06, \text{memtol}=1e-06, \text{bose}=\text{True}, \text{tasks}=0) \]

Applies the homotopy membership test for a point to belong to a witness set of dimension \( \text{dim} \), given by an embedding polynomial Laurent system in \( \text{wsys} \), with corresponding generic points in \( \text{gpts} \). The coordinates of the test point are given in the string \( \text{point} \), which is the string representation of a solution in PHCpack format, with symbols of the variables before the values of the coordinates. By default, \( \text{verbose} \) is True. The number of threads is given in \( \text{tasks} \). If \( \text{tasks} \) is zero, then no multithreading is applied in the homotopy membership test. Calculations happen in standard double precision. The default values for the evaluation (\( \text{evatol} \)) and the membership (\( \text{memtol} \)) allow for singular values at the end points of the paths in the homotopy membership test. Returns a tuple of two booleans. The first boolean is True if the point satisfies the equations, otherwise it is False. The second boolean is True if the point belongs to the witness set, otherwise, the second boolean is False.

\[ \text{sets.standard_laurent_ismember_filter}(\text{wsys}, \text{gpts}, \text{dim}, \text{points}, \text{rcotol}=1e-06, \text{evatol}=1e-06, \text{memtol}=1e-06, \text{bose}=\text{True}, \text{tasks}=0) \]

Given in \( \text{wsys} \) and \( \text{gpts} \) is a witness set of dimension \( \text{dim} \), where \( \text{wsys} \) is an embedded Laurent polynomial system, and in \( \text{points} \) a list of strings. The strings represent points as solutions in PHCpack format. The homotopy membership test is applied to each point in the list \( \text{points} \). The list on return contains the points that do NOT belong to the witness set. Points that belong to the witness set are considered junk. By default, \( \text{verbose} \) is True. The number of threads is given in \( \text{tasks} \). If \( \text{tasks} \) is zero, then no multithreading is applied in the homotopy membership test. Calculations happen in standard double precision. The parameter \( \text{rcotol} \) is used to bypass the homotopy membership test, for points with their estimated inverse condition number larger than \( \text{rcotol} \) will be considered isolated and not in the witness set.

\[ \text{sets.standard_laurent_membertest}(\text{wsys}, \text{gpts}, \text{dim}, \text{point}, \text{evatol}=1e-06, \text{memtol}=1e-06, \text{bose}=\text{True}, \text{tasks}=0) \]

Applies the homotopy membership test for a point to belong to a witness set of dimension \( \text{dim} \), given by an embedding Laurent system in \( \text{wsys} \), with corresponding generic points in \( \text{gpts} \). The coordinates of the test point are given in the list \( \text{point} \), as a list of doubles, with the real and imaginary part of each coordinate of the point. By default, \( \text{verbose} \) is True. The number of threads is given in \( \text{tasks} \). If \( \text{tasks} \) is zero, then no multithreading is applied in the homotopy membership test. Calculations happen in standard 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.

```python
sets.standard_membertest(wsys, gpts, dim, point, evatol=1e-06, memtol=1e-06, verbose=True, tasks=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 standard 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.

```python
sets.test()
```

Runs a test on algebraic sets.

```python
sets.test_member(prc='d')
```

To test the membership, we take the twisted cubic.

```python
sets.test_twisted_ismember(prc='d', laurent=True)
```

To test the membertest wrapper, we take the twisted cubic again. The test point is given as a solution in PHCpack format.

```python
sets.witness_set_of_hypersurface(nvar, hpol, precision='d')
```

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 \( hpol \). Three different precisions are supported, by default double (‘d’), or otherwise double double (‘dd’) or quad double (‘qd’).

```python
sets.witness_set_of_laurent_hypersurface(nvar, hpol, precision='d')
```

Given in \( hpol \) the string representation of a laurent 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 may differ from the actual degree of \( hpol \) if the polynomial represented by \( hpol \) has negative exponents. Three different precisions are supported, by default double (‘d’), or otherwise double double (‘dd’) or quad double (‘qd’).

### 4.3.2 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.

```python
cascades.cascade_filter(dim, embpols, nonsols, tol, nbtasks=0, prc='d', verbose=True)
```

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. By default, the precision \( prc \) is double (‘d’). Other valid values for \( prc \) are ‘dd’ (for double double) and ‘qd’ (for quad double). If \( verbose \), then some output is written to screen.

```python
cascades.cascade_step(dim, embsys, esols, precision='d', tasks=0)
```

Given in \( embsys \) an embedded polynomial system and solutions with nonzero slack variables in \( esols \), does one step in the homotopy cascade, with \( precision \)

- \( d \): 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)).

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_step**(dim, embsys, esols, tasks=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_step**(dim, embsys, esols, tasks=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.laurent_cascade_filter**(dim, embpols, nonsols, tol, nbtasks=0, prc='d', verbose=True)

Runs one step in the cascade homotopy defined by the embedding of Laurent 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. By default, the precision `prc` is double (‘d’). Other valid values for `prc` are ‘dd’ (for double double) and ‘qd’ (for quad double). If `verbose`, then some output is written to screen.

**cascades.laurent_cascade_step**(dim, embsys, esols, precision='d', tasks=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 precision

- `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)).

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.laurent_top_cascade**(nvr, dim, pols, tol, nbtasks=0, prc='d', verbose=True)

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. The tolerance `tol` is used to split the solution list in the list of 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 are

1. `nbtasks` is the number of tasks, 0 if no multitasking;
2. the working precision `prc`, ‘d’ for double, ‘dd’ for double double, or ‘qd’ for quad double;
3. if `verbose`, then some output is written to screen.

**cascades.quad_double_cascade_step**(dim, embsys, esols, tasks=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_step(dim, embsys, esols, tasks=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.run_cascade(nvr, dim, polys, islaurent=False, tol=1e-08, rcotol=1e-06, evatol=1e-06, memtol=1e-06, tasks=0, prc='d', verbose=True)

Runs a cascade on the polynomials polys, in the number of variables equal to nvr, starting at the top dimension dim. If islaurent, then the polynomials in polys may have negative exponents. Returns a dictionary with as keys the dimensions and as values the tuples with the embedded systems and the corresponding generic points. Four tolerance parameters have default values on input: tol is used to decide which slack variables are zero, rcotol is the tolerance on the estimated inverse condition number, evatol is the tolerance on the residual to filter junk points, memtol is the tolerance for the homotopy membership test. The number of tasks is given by tasks (0 for no multitasking) and the default precision is double. Other supported values for prc are ‘dd’ for double double and ‘qd’ for quad double. If verbose, then a summary of the filtering is printed.

cascades.split_filter(sols, dim, tol, verbose=True)

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 verbose, then the length of each solution list is printed.

cascades.standard_double_cascade_step(dim, embsys, esols, tasks=0)

Given in embsys an embedded polynomial system and solutions with nonzero slack variables in esols, does one step in the homotopy cascade, with standard 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.standard_double_laurent_cascade_step(dim, embsys, esols, tasks=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 standard 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.test()

Fixes a seed for the random number generators before running the test on the cascade homotopies.

cascades.test_cascade()

Does one cascade step on simple example. In the top embedding we first find the 2-dimensional solution set $x = 1$. In the cascade step we compute the three witness points on the twisted cubic.

cascades.test_run_cascade()

Runs the cascade on a list of polynomials.

cascades.top_cascade(nvr, dim, polys, tol, nbtasks=0, prc='d', verbose=True)

Constructs an embedding of the polynomials in polys, with the number of variables in polys equal to nvr, where dim is the top dimension of the solution set. Applies the blackbox solver to the embedded system. The tolerance tol is used to split the solution list in the list of 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.

4.3. positive dimensional solution sets

125
The three parameters are
1. \texttt{nbtasks} is the number of tasks, 0 if no multitasking;
2. the working precision \texttt{prc}, ‘d’ for double, ‘dd’ for double double, or ‘qd’ for quad double;
3. if \texttt{verbose}, then some output is written to screen.

### 4.3.3 functions in the module \texttt{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.

\texttt{factor.decompose (deco, islaurent=0, verbose=True, nbloops=20, precision=’d’)}
Given in \texttt{deco} is a dictionary with as keys the dimension and as value a tuple with an embedded (Laurent) polynomial system and its corresponding solutions as the second item in the tuple. Each item in the dictionary is decomposed into irreducible factors. If the embedded polynomial system is a Laurent system, then islaurent must equal one, the default is zero. The default precision is double ‘d’. Other valid values for precision are ‘dd’ for double double, or ‘qd’ for quad double. Returns the dictionary deco, but each tuple (except for dimension 0) is extended with the partition of the generic points with the linear trace difference to represented the irreducible decomposition.

\texttt{factor.decomposition (deg, precision=’d’)}
Returns the decomposition as a list of labels of witness points on the components, computed in precision ‘d’, ‘dd’, or ‘qd’, respectively for double, double double, or quad double.

\texttt{factor.dobldobl_decomposition (deg)}
Returns the decomposition as a list of labels of witness points on the components, computed in double double precision.

\texttt{factor.dobldobl_laursys_solve (pols, topdim=-1, filter=True, factor=True, tasks=0, verbose=True)}
Runs the cascades of homotopies on the Laurent polynomial system in pols in double double precision. The default top dimension topdim is the number of variables in pols minus one.

\texttt{factor.dobldobl_monodromy_breakup (embsys, esols, dim, islaurent=0, verbose=True, nbloops=0)}
Applies the monodromy breakup algorithm in double double precision to factor the \texttt{dim}-dimensional algebraic set represented by the embedded system \texttt{embsys} and its solutions \texttt{esols}. If the embedded polynomial system is a Laurent system, then islaurent must equal one, the default is zero. If \texttt{verbose} is False, then no output is written. If \texttt{nbloops} equals zero, then the user is prompted to give the maximum number of loops.

\texttt{factor.dobldobl_polysys_solve (pols, topdim=-1, filter=True, factor=True, tasks=0, verbose=True)}
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.

\texttt{factor.factor (dim, witsys, witsols, islaurent=0, verbose=True, nbloops=20, precision=’d’)}
Applies monodromy to factor an equidimensional algebraic set, given as a witness sets, with the embedded polynomials in \texttt{witsys}, and corresponding generic points in \texttt{witsols}. If the embedded polynomial system is a Laurent system, then islaurent must equal one, the default is zero. The dimension of the algebraic set is given in \texttt{dim}. The default precision is double ‘d’. Other valid values for precision are ‘dd’ for double double, or ‘qd’ for quad double.

\texttt{factor.laursys_solve (pols, topdim=-1, precision=’d’, filter=True, factor=True, tasks=0, verbose=True)}
Runs the cascades of homotopies on the Laurent polynomial system in pols in double, double double, or quad double precision. The default top dimension topdim is the number of variables in pols minus one.

\texttt{factor.monodromy_breakup (embsys, esols, dim, islaurent=0, verbose=True, nbloops=0, prec=’d’)}
Applies the monodromy breakup algorithm to factor the \texttt{dim}-dimensional set represented by the embedded
factor.\textbf{polysys\_solve}(pols, topdim=-1, precision='d', filter=True, factor=True, tasks=0, verbose=True)

Runs the cascades of homotopies on the polynomial system in \texttt{pols} in double, double double, or quad double precision. The default top dimension \texttt{topdim} is the number of variables in \texttt{pols} minus one.

factor.\textbf{quaddobl\_decomposition}(deg)

Returns the decomposition as a list of labels of witness points on the components, computed in quad double precision.

factor.\textbf{quaddobl\_laursys\_solve}(pols, topdim=-1, filter=True, factor=True, tasks=0, verbose=True)

Runs the cascades of homotopies on the Laurent polynomial system in \texttt{pols} in quad double precision. The default top dimension \texttt{topdim} is the number of variables in \texttt{pols} minus one.

factor.\textbf{quaddobl\_monodromy\_breakup}(embsys, esols, dim, islaurent=0, verbose=True, nbloops=0)

Applies the monodromy breakup algorithm in quad double precision to factor the \texttt{dim}-dimensional algebraic set represented by the embedded system \texttt{embsys} and its solutions \texttt{esols}. If the embedded polynomial system is a Laurent system, then \texttt{islaurent} must equal one, the default is zero. If \texttt{verbose} is False, then no output is written. If \texttt{nbloops} equals zero, then the user is prompted to give the maximum number of loops.

factor.\textbf{quaddobl\_polysys\_solve}(pols, topdim=-1, filter=True, factor=True, tasks=0, verbose=True)

Runs the cascades of homotopies on the polynomial system in \texttt{pols} in quad double precision. The default top dimension \texttt{topdim} is the number of variables in \texttt{pols} minus one.

factor.\textbf{solve}(nvr, dim, pols, islaurent=False, precision='d', tasks=0, nbloops=20, tol=1e-08, rcotol=1e-06, evatol=1e-06, memtol=1e-06, verbose=True)

Computes a numerical irreducible decomposition for the polynomials in the list \texttt{pols}, where \texttt{nvr} is the number of variables in \texttt{pols}. The top dimension (the highest dimension of the solution set) is given in \texttt{dim} and could be by default set to \texttt{nvr}-1. If \texttt{islaurent}, then \texttt{pols} is considered a Laurent polynomial system and negative exponents may occur. The default precision is double ‘d’. Other valid values for precision are ‘dd’ for double double, or ‘qd’ for quad double. On return is a dictionary. The keys in the dictionary are dimensions. For each dimension, a tuple represents a witness set. For dimension zero, the solution list contains the isolated solutions. For each nonzero dimension, the generic points in the witness set are partitioned according to the irreducible factors of the solution set at that dimension.

factor.\textbf{standard\_decomposition}(deg)

Returns the decomposition as a list of labels of witness points on the components, computed in standard double double precision.

factor.\textbf{standard\_laursys\_solve}(pols, topdim=-1, filter=True, factor=True, tasks=0, verbose=True)

Runs the cascades of homotopies on the Laurent polynomial system in \texttt{pols} in standard double precision. The default top dimension \texttt{topdim} is the number of variables in \texttt{pols} minus one.

factor.\textbf{standard\_monodromy\_breakup}(embsys, esols, dim, islaurent=0, verbose=True, nbloops=0)

Applies the monodromy breakup algorithm in standard double precision to factor the \texttt{dim}-dimensional algebraic set represented by the embedded system \texttt{embsys} and its solutions \texttt{esols}. If the embedded polynomial system is a Laurent system, then \texttt{islaurent} must equal one, the default is zero. If \texttt{verbose} is False, then no output is written. If \texttt{nbloops} equals zero, then the user is prompted to give the maximum number of loops.

factor.\textbf{standard\_polysys\_solve}(pols, topdim=-1, filter=True, factor=True, tasks=0, verbose=True)

Runs the cascades of homotopies on the polynomial system in \texttt{pols} in standard double precision. The default
top dimension \( \text{topdim} \) is the number of variables in \( \text{pols} \) minus one.

```python
def factor.test():
    """Sets the seed for the random number generators to a fixed number and then runs a test."""
    return

def factor.test_decompose():
    """Runs a test on the \text{decompose()} function."""
    return

def factor.test_factor():
    """Simple test on the factor method."""
    return

def factor.test_monodromy(prc='d'):
    """Runs a test on applying \text{monodromy loops} to factor a curve into irreducible components."""
    return

def factor.test_polysys_solve():
    """Runs a test on the \text{standard\_polysys\_solve()} function."""
    return

def factor.test_solve():
    """Runs a test on the solve() function."""
    return

def factor.write_decomposition(deco):
    """Given in \text{deco} is a dictionary where the keys are dimensions. For each dimension, there is a tuple with a witness set representation of the solution set at that dimension. The decomposition in \text{deco} is written."""
    return
```

### 4.3.4 functions in the module \text{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.

```python
def \text{diagonal\_solver}(dim, dm1, sys1, sols1, dm2, sys2, sols2, tasks=0, prc='d', verbose=True):
    """Runs the diagonal homotopies to intersect two witness sets stored in \((sys1, sols1)\) and \((sys2, sols2)\), of respective dimensions \(dim1\) and \(dim2\). The ambient dimension equals \(dim\). Multitasking is available, and is activated by the \text{tasks} parameter. The precision is set by the parameter \text{prc}, which takes the default value \('d'\) for standard double, \('dd'\) for double double, or \('qd'\) for quad double precision. Returns the last system in the cascade and its solutions."
    return
```

```python
def \text{dobldobl\_diagonal\_cascade\_solutions}(dim1, dim2):
    """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, \text{dobldobl\_diagonal\_homotopy} must have been executed successfully."
    return
```

```python
def \text{dobldobl\_diagonal\_homotopy}(dim1, sys1, esols1, dim2, sys2, esols2):
    """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."
    return
```

```python
def \text{dobldobl\_diagonal\_solver}(dim, dm1, sys1, sols1, dm2, sys2, sols2, tasks=0, verbose=True):
    """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 \text{tasks} parameter. Returns the last system in the cascade and its solutions. If \text{verbose}, then the solver runs in interactive mode, printing intermediate results to screen and prompting the user to continue."
    return
```

```python
def \text{dobldobl\_start\_diagonal\_cascade}(gamma=0, tasks=0):
    """Does the path tracking to start a diagonal cascade in double double precision. For this to work, the functions \text{dobldobl\_diagonal\_homotopy} and \text{dobldobl\_diagonal\_cascade\_solutions} must be executed successfully. If \text{gamma} equals \(0\) on input, then a random gamma constant is generated, otherwise, the given complex gamma"
    return
```
diagonal.\texttt{quaddobl_diagonal_cascade_solutions}(\texttt{dim1}, \texttt{dim2})

Defines the start solutions in the cascade to start the diagonal homotopy to intersect a set of dimension \texttt{dim1} with another set of dimension \texttt{dim2}, in quad double precision. For this to work, \texttt{quaddobl_diagonal_homotopy} must have been executed successfully.

\texttt{diagonal.quaddobl_diagonal_homotopy}(\texttt{dim1}, \texttt{sys1}, \texttt{esols1}, \texttt{dim2}, \texttt{sys2}, \texttt{esols2})

Defines a diagonal homotopy to intersect the witness sets defined by \texttt{(sys1, esols1)} and \texttt{(sys2, esols2)}, respectively of dimensions \texttt{dim1} and \texttt{dim2}. The systems \texttt{sys1} and \texttt{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.

\texttt{diagonal.quaddobl_diagonal_solver}(\texttt{dim}, \texttt{dm1}, \texttt{sys1}, \texttt{sols1}, \texttt{dm2}, \texttt{sys2}, \texttt{sols2}, \texttt{tasks=0, verbose=True})

Runs the diagonal homotopies in quad double precision to intersect two witness sets stored in \texttt{(sys1, sols1)} and \texttt{(sys2, sols2)}, of respective dimensions \texttt{dim1} and \texttt{dim2}. The ambient dimension equals \texttt{dim}. Multitasking is available, and is activated by the \texttt{tasks} parameter. Returns the last system in the cascade and its solutions. If \texttt{verbose}, then the solver runs in interactive mode, printing intermediate results to screen and prompting the user to continue.

\texttt{diagonal.quaddobl_start_diagonal_cascade}(\texttt{gamma=0, tasks=0})

Does the path tracking to start a diagonal cascade in quad double precision. For this to work, the functions \texttt{quaddobl_diagonal_homotopy} and \texttt{quaddobl_diagonal_cascade_solutions} must be executed successfully. If \texttt{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 is activated by the \texttt{tasks} parameter. Returns the target (system and its corresponding) solutions.

\texttt{diagonal.standard_diagonal_cascade_solutions}(\texttt{dim1}, \texttt{dim2})

Defines the start solutions in the cascade to start the diagonal homotopy to intersect a set of dimension \texttt{dim1} with another set of dimension \texttt{dim2}, in standard double precision. For this to work, \texttt{standard_diagonal_homotopy} must have been executed successfully.

\texttt{diagonal.standard_diagonal_homotopy}(\texttt{dim1}, \texttt{sys1}, \texttt{esols1}, \texttt{dim2}, \texttt{sys2}, \texttt{esols2})

Defines a diagonal homotopy to intersect the witness sets defined by \texttt{(sys1, esols1)} and \texttt{(sys2, esols2)}, respectively of dimensions \texttt{dim1} and \texttt{dim2}. The systems \texttt{sys1} and \texttt{sys2} are assumed to be square and with as many slack variables as the dimension of the solution sets. The data is stored in standard double precision.

\texttt{diagonal.standard_diagonal_solver}(\texttt{dim}, \texttt{dm1}, \texttt{sys1}, \texttt{sols1}, \texttt{dm2}, \texttt{sys2}, \texttt{sols2}, \texttt{tasks=0, verbose=True})

Runs the diagonal homotopies in standard double precision to intersect two witness sets stored in \texttt{(sys1, sols1)} and \texttt{(sys2, sols2)}, of respective dimensions \texttt{dm1} and \texttt{dm2}. The ambient dimension equals \texttt{dm}. Multitasking is available, and is activated by the \texttt{tasks} parameter. Returns the last system in the cascade and its solutions. If \texttt{verbose}, then the solver runs in interactive mode, printing intermediate results to screen and prompting the user to continue.

\texttt{diagonal.standard_start_diagonal_cascade}(\texttt{gamma=0, tasks=0})

Does the path tracking to start a diagonal cascade in standard double precision. For this to work, the functions \texttt{standard_diagonal_homotopy} and \texttt{standard_diagonal_cascade_solutions} must be executed successfully. If \texttt{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 \texttt{tasks} parameter. Returns the target (system and its corresponding) solutions.

\texttt{diagonal.test()}

Runs a test on algebraic sets.

\texttt{diagonal.test_diaghom}(\texttt{precision='d'})

Test on the diagonal homotopy.

4.3. \textit{positive dimensional solution sets}
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.

4.4 some interesting families and examples

One of the motivations for phcpy was to perform regression tests on the blackbox solver.

4.4.1 functions in the module examples

The documentation strings of the functions that return the polynomials of the example systems as strings of characters are listed below. The regression test is exported by the function test() of the module examples.

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 test 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()

examples.fbrfive4()

examples.game4two()

examples.katsura6()

examples.noon3()
4.4. some interesting families and examples

examples.rps10()

examples.sevenbar()

examples.solve_binomials()
    Runs the test on solving the binomials example. Asserts that the number of solutions equals 20.

examples.solve_cyclic7()
    Runs the test on solving the cyclic 7-roots problem. Asserts that the number of solutions equals 924.

examples.solve_fbrfive4()
    Runs the test on solving a generic 4-bar problem. Asserts that the number of solutions equals 36.

examples.solve_game4two()
    Runs the test on solving the Nash equilibrium problem. Asserts that the number of solutions equals 9.

examples.solve_katsura6()
    Runs the test on solving the katsura6 problem. Asserts that the number of solutions equals 64.

examples.solve_noon3()
    Test on solving the noon3 system. Asserts that the number of solutions equals 21.

examples.solve_rps10()
    Test on solving a mechanical design problem. Asserts that the number of solutions equals 1024.

examples.solve_sevenbar()
    Test on solving a special 7-bar problem. Asserts that the number of isolated solutions equals 6 and that there are three generic points at a curve.

examples.solve_stewgou40()
    Test on solving a fully real Stewart-Gough platform. Asserts that the number of solutions equals 40.

examples.solve_sysd1()
    Runs the test on solving the benchmark problem D1. Asserts that the number of solutions equals 48.

examples.solve_tangents()
    Test on solving the tangents to 4 spheres problem. Asserts that the number of solutions equals 6.

examples.stewgou40()

examples.sysd1()

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 tetrahedron. See the paper by Frank Sottile: From Enumerative Geometry to Solving Systems of Polynomial Equations". In “Computations in Algebraic Geometry with Macaulay 2”,

31 131
include examples.test()
Solves the systems and tests on their number of solutions.

4.4.2 functions in the module families

The module families contains scripts to generate polynomial systems for any dimension.

families.adjacent_minors(rows, cols)

families.chandra(dim, par=0.51234)

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)

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 equilibria for nbplayers with two pure strategies, with random positive utilities.

families.nbdeqs(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)

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.
4.5 numerical Schubert calculus

The module schubert exports Pieri homotopies and Littlewood-Richardson homotopies to solve Schubert problems.

4.5.1 functions in the module schubert

PHCpack offers numerical Schubert calculus, exported here.

\begin{description}
\item[schubert.cheater] \textit{(mdim, pdim, qdeg, start, startsols)}
\begin{itemize}
\item Generates a random Pieri problem of dimensions \textit{(mdim, pdim, qdeg)} and solves it with a Cheater's homotopy, starting from the Pieri system in \textit{start}, at the solutions in \textit{startsols}.
\end{itemize}
\item[schubert.dobldobl_littlewood_richardson_homotopies] \textit{(ndim, kdim, brackets, verbose=True, vrfcnd=False, minrep=True, tosqr=False, outputfilename='')} 
\begin{itemize}
\item In n-dimensional space we consider k-dimensional planes, subject to intersection conditions represented by brackets. The parameters \textit{ndim} and \textit{kdim} give values for n and k respectively. The parameter \textit{brackets} is a list of brackets. A bracket is a list of as many natural numbers (in the range 1..*ndim*) as \textit{kdim}. The Littlewood-Richardson homotopies compute k-planes that meet the flags at spaces of dimensions prescribed by the brackets, in double double precision. Four options are passed as Booleans:
\item[\textit{verbose}]: for adding extra output during computations,
\item[\textit{vrfcnd}]: for extra diagnostic verification of Schubert conditions,
\item[\textit{minrep}]: for a minimal representation of the problem formulation,
\item[\textit{tosqr}]: to square the overdetermined systems.
\end{itemize}
\item[schubert.littlewood_richardson_homotopies] \textit{(ndim, kdim, brackets, verbose=True, vrfcnd=False, minrep=True, tosqr=False, precision='d', outputfilename='')} 
\begin{itemize}
\item In n-dimensional space we consider k-dimensional planes, subject to intersection conditions represented by brackets. The parameters \textit{ndim} and \textit{kdim} give values for n and k respectively. The parameter \textit{brackets} is a list of brackets. A bracket is a list of as many natural numbers (in the range 1..*ndim*) as \textit{kdim}. The Littlewood-Richardson homotopies compute k-planes that meet the flags at spaces of dimensions prescribed by the brackets. Four options are passed as Booleans:
\item[\textit{verbose}]: for adding extra output during computations,
\item[\textit{vrfcnd}]: for extra diagnostic verification of Schubert conditions,
\item[\textit{minrep}]: for a minimal representation of the problem formulation,
\item[\textit{tosqr}]: to square the overdetermined systems.
\end{itemize}
\item[schubert.main] 
\begin{itemize}
\item Tests the Pieri homotopies and the Littlewood-Richardson homotopies.
\end{itemize}
\end{description}
schubert.make_pieri_system(mdim, pdim, qdeg, planes, is_real=False)
    Makes the polynomial system defined by the mdim-planes in the list planes.

schubert.osculating_input(mdim, pdim, qdeg, start, startsols)
    Generates real mdim-planes osculating a rational normal curve and solves this Pieri problem using the system in start, with corresponding solutions in startsols.

schubert.pieri_root_count(mdim, pdim, qdeg, verbose=True)
    Computes the number of pdim-plane producing maps of degree qdeg that meet mdim-planes at mdim*pdim + qdeg*(mdim+pdim) points.

schubert.planes_to_string(planes)
    Returns one long string with all numbers in planes, a list of lists of rows. The numbers are the real and imaginary parts, separated by space.

schubert.points_to_string(pts)
    Returns one long string with all numbers in pts, as sequences of real and imaginary parts, every number is separated by one space.

schubert.prompt_for_dimensions()
    Returns the triplet (m,p,q), where m is the dimension of the input planes, p is the dimension of the output planes, and q is the degree of the maps.

schubert.quaddobl_littlewood_richardson_homotopies(ndim, kdim, brackets, verbose=True, vrfcnd=False, minrep=True, tosqr=False, outputfilename='')
    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 quad 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 minimal 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, and the coordinates of the flags, 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.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)
    Returns m*p + qdeg*(m+p) real m-planes osculating a rational normal curve.

schubert.resolve_schubert_conditions(ndim, kdim, brackets, verbose=True)
    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, *pts, **opt)
Computes the number of pdim-plane producing maps of degree qdeg that meet mdim-planes at mdim*pdim + qdeg*(mdim+pdim) points. For qdeg = 0, there are no interpolation points in pts.

schubert.standard_littlewood_richardson_homotopies(ndim, kdim, brackets, verbose=True, vrfcnd=False, minrep=True, tosqr=False, outputfilename='')
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 minimal 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.test_lrhom(prc='d')
Performs a test on the Littlewood-Richardson homotopies.

schubert.test_pieri()
Does a test on the Pieri homotopies.

schubert.verify(pols, sols)
Verifies whether the solutions in sols satisfy the polynomials of the system in pols.

### 4.6 Newton polytopes, monomial maps, and power series

The Newton polytope of a polynomial is spanned by the exponents of monomials which occur with nonzero coefficient in the polynomial.

#### 4.6.1 functions in the module polytopes

Given a polynomial, its support is the set of exponents of monomials which occur with nonzero coefficient. The convex hull of the support of a polynomial is the Newton polytope of the polynomial. For a polynomial system, the mixed volume of the Newton polytopes of the polynomials in the systems gives a generically sharp upper bound on the number of isolated solutions (not in coordinate planes) of the polynomial system.

This module exports routines of PHCpack to work with Newton polytopes.

topotes.check_mixture(mixture, points)
The sum of the integers in the list mixture equal the dimension of each point in points. Returns True if the mixture type passes the test, otherwise, prints an error message and returns False.

topotes.convex_hull(dim, points, checkin=True, checkout=True)
Returns the list of facets of the convex hull of the points, given in points. The dimension of the ambient space is in dim. If checkin (by default), the type of the input is checked. If checkout (by default), the output is checked.
polytopes.convex_hull_checkin \((\text{dim}, \text{points})\)
Checks whether the input arguments satisfy the requirements: \text{points} is a list of tuples that each contain as many integer numbers as the value of \text{dim}. Returns True if the requirements are satisfied, returns False otherwise.

polytopes.convex_hull_checkout \((\text{dim}, \text{points}, \text{facets}, \text{verbose}=\text{True})\)
Checks whether for each facet in the list of \text{facets}, the facet is supported on the \text{points} defined by the computed inner normal and the minimal value. Returns True if the check passes, returns False otherwise.

polytopes.edges_in_facets \((\text{facets})\)
Given the the list of \text{facets}, returns the list of tuples of indices to the point that span the edges of the facets.

polytopes.initial_form \((\text{pols}, \text{normal})\)
Returns the initial form of the polynomials in \text{pols} with respect to the inner normal with coordinates in \text{normal}.

polytopes.initial_support \((\text{points}, \text{normal})\)
Returns the list of elements in \text{points} that make the minimal inner product with the given \text{normal}, as the second element in a tuple. The first element is the value of that minimal inner product. Every tuple in \text{points} must have the same length as \text{normal}.

polytopes.integer_mixed_cell \((\text{dim}, \text{nbr}, \text{idx}, \text{verbose}=\text{True})\)
Given are three integers and one boolean, respectively:
- \text{dim}: the number of coordinates in the inner normal,
- \text{nbr}: the number of distinct supports,
- \text{idx}: the index to the cell (starts at one, instead of at zero), and
- \text{verbose}: the verbose flag.

Returns the extracted data for the mixed cell with index \text{idx}. If verbose, the data is written to screen.

polytopes.integer_mixed_cells \((\text{mixture}, \text{points}, \text{verbose}=\text{True})\)
Given a tuple of lifted support sets in \text{points}, computes all mixed cells in the regular subdivision defined by the integer lifting values given as the last coordinate of every point in the lifted supports. If \text{verbose}, then output is written to screen. Returns the mixed volume as the sum of the volumes of the cells.

polytopes.mixed_volume \((\text{mixture}, \text{points}, \text{checkin}=\text{True})\)
Returns the mixed volume of the list of lists in \text{points}. Both \text{mixture} and \text{points} have the same length. The list \text{mixture} counts the number of times each support in \text{points} should be counted. For example, to compute the volume of a three dimensional polytope, the mixture is [3]. In general, the mixture determines the powers of the unknowns in the Minkowski polynomial of which the computed mixed volume is its coefficient. If checkin, then the mixture will be tested to match the length of each point in points. Examples: >>> q1 = [(1, 1), (1, 0), (0, 1), (0, 0)] >>> q2 = [(2, 2), (1, 0), (0, 1)] >>> mv([1, 1], [q1, q2]) 4 >>> mv([2], [q1]) 2

polytopes.planar_convex_hull \((\text{points}, \text{checkin}=\text{True}, \text{checkout}=\text{True})\)
The convex hull of a point configuration in the plane consists of an ordered list of vertex \text{points}, ordered such that any two consecutive points span an edge, with the list of corresponding inner normals. If \text{checkin} (by default), the type of the input is checked. If \text{checkout} (by default), the output is checked.

polytopes.planar_hull_checkout \((\text{vertices}, \text{normals}, \text{verbose}=\text{True})\)
Given a list of \text{vertices} and a list of \text{normals} as output of a convex hull algorithm in the plane, this function checks whether the initial support of every normal consists of exactly two points that appear with consecutive indices (modulo the length of the list) in the list of \text{vertices}. Return True if the checks pass, False otherwise.

polytopes.random_points \((\text{dim}, \text{nbr}, \text{low}, \text{upp})\)
Generates a list of random integer points. Returns a list of \text{nbr} points of dimension \text{dim}, with integer coordinates in the range from \text{low} to \text{upp}.

polytopes.support \((\text{nvr}, \text{pol})\)
The support of a multivariate polynomial is a set of exponents of the monomials that appear with nonzero
coefficient. Given in $nvr$ the number of variables and in $pol$ a string representation of a polynomial in $nvr$ variables, returns the support of the polynomial as a list of tuples.

polytopes.*support_sets*($pols$, **$nbvar$)
Returns the support sets of all polynomials in the list $pols$. The $nbvar$ is the number of variables in the system. if $nbvar$ is omitted, then the system is assumed to be square.

polytopes.*test_convex_hull*($dim=3$, $nbr=10$, $size=9$)
Generates a random point configuration in 3-space by default (although also $dim = 4$ works) and then computes its convex hull. By default, 10 points are generated, while in general, the number of points in the configurations equals $nbr$. The range of the coordinates in the point is defined by the value of $size$, from -$size$ to $size$.

polytopes.*test_integer_mixed_volume*()
Tests mixed volume computation via integer valued lifting functions.

polytopes.*test_mixed_volume*()
Runs some simple tests on mixed volume computation.

polytopes.*test_planar_hull*($nbr=7$, $size=9$)
Generates a random point configuration in the plane and then computes its convex hull. By default, the number of points equals 7, in general it is the value of the parameter $nbr$. The range of the coordinates in the point is defined by the value of $size$, from -$size$ to $size$.

polytopes.*vertices_in_facets*($facets$)
Given the list of $facets$, returns the list of indices to the vertices, to the points that span the facets.

4.6.2 functions in the module maps

A binomial system is a system where every equation has exactly two monomials with nonzero coefficient. The solution set of a binomial system is a set of monomial maps.

This module allows to work with monomial maps, defined by binomial systems.

maps.*is_binomial_system*($silent=True$)
Returns True if the system stored in the Laurent systems container is a binomial system, returns False otherwise. if not $silent$, then the number of terms in each Laurent polynomial is written to screen.

maps.*monomial_map_solutions*($nbvar$, $with\_degree=True$)
Returns the list of lists of strings, each list of strings representing a monomial map stored in the container. The number of variables equals $nbvar$.

maps.*monomial_map_strings*($dim$, $ind$, $nbvar$)
Returns the list of strings representing the components of the monomial map of dimension $dim$, with index $ind$, and where the number of variables equals $nbvar$.

maps.*solve_binomials*($nbvar$, $pols$, $silent=True$, $puretopdim=False$)
If the system given in $pols$ as a list of strings in as many variables as the value of $nbvar$ is a binomial system (that is: it has exactly two monomials with a nonzero coefficient in every equation), then this function will return monomial maps to represent the solution sets. By default, $silent$ is True and no additional output is written. If only the expected pure top dimensional solution sets are of interest, then switch the default $puretopdim$ to True for faster results. The expected top dimension equals the number of variables minus the number of equations.

maps.*store_laurent_system*($nbvar$, $pols$)
Given in $pols$ a list of string representing Laurent polynomials into the systems container. The number of variables equals $nbvar$.

maps.*test*()
Solves a binomial system which has the x-axis, the yz-plane, and the twisted cubic as solution components. The yz-plane is a solution set of the unexpected dimension 2.
maps.write_monomial_map(dim, ind, nbvar)
Write the monomial map of dimension dim and of index ind, with number of variables equal to nbvar.

maps.write_monomial_maps(nbvar)
Writes the maps stored in the container. The number of variables is given in nbvar.

4.6.3 functions in the module series

Newton’s method over the field of truncated power series computes series expansions for solution curves.

The module series exports functions to compute power series solutions with Newton’s method in double, double double, or quad double precision.

series.apollonius(precision='d')
Test on computing the power series at a double solution for the problem of Apolonius. The parameter t is the fourth variable, whence we call Newton’s method with idx equal to four.

series.checkin_newton_power_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.dobldobl_newton_power_series(pols, lser, idx=1, maxdeg=4, nbr=4, checkin=True, verbose=True)
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.
- verbose: whether to write intermediate output to screen or not.

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.dobldobl_newton_series(pols, sols, idx=1, maxdeg=4, nbr=4, verbose=True)
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 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,
- verbose: whether to write intermediate output to screen or not.

On return is a list of lists of strings. Each lists of strings represents the series solution for the variables in the list pols.
Computes Pade approximants based on the series in double double precision for the polynomials in \texttt{pols}, where the leading coefficients of the series are the solutions in \texttt{sols}. On entry are the following seven parameters:

- \texttt{pols}: a list of string representations of polynomials,
- \texttt{sols}: a list of solutions of the polynomials in \texttt{pols},
- \texttt{idx}: index of the series parameter, by default equals 1,
- \texttt{numdeg}: the degree of the numerator,
- \texttt{dendeg}: the degree of the denominator,
- \texttt{nbr}: number of steps with Newton’s method,
- \texttt{verbose}: whether to write intermediate output to screen or not.

On return is a list of lists of strings. Each lists of strings represents the series solution for the variables in the list \texttt{pols}.

The function \( f(z) = ((1 + 1/2*z)/(1 + 2*z))^\frac{1}{2} \) is a solution \( x(s) \) of \( (1-s) \cdot (x^2 - 1) + s \cdot (3 \cdot x^2 - 3/2) = 0 \)

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.

Computes series in quad double precision for the polynomials in \texttt{pols}, where the leading terms are given in the list \texttt{lser}. On entry are the following five parameters:

- \texttt{pols}: a list of string representations of polynomials,
- \texttt{lser}: a list of polynomials in the series parameter (e.g.: \texttt{t}), for use as start terms in Newton’s method,
- \texttt{idx}: index of the series parameter, by default equals 1,
- \texttt{maxdeg}: maximal degree of the series,
- \texttt{nbr}: number of steps with Newton’s method,
- \texttt{checkin}: checks whether the number of symbols in \texttt{pols} matches the length of the list \texttt{lser} if \texttt{idx} == 0, or is one less than the length of the list \texttt{lser} if \texttt{idx} != 0. If the conditions are not satisfied, then an error message is printed and \texttt{lser} is returned.
- \texttt{verbose}: whether to write intermediate output to screen or not.

On return is a list of lists of strings. Each lists of strings represents the series solution for the variables in the list \texttt{pols}.

Computes series in quad double precision for the polynomials in \texttt{pols}, where the leading coefficients are the solutions in \texttt{sols}. On entry are the following five parameters:

- \texttt{pols}: a list of string representations of polynomials,
- \texttt{sols}: a list of solutions of the polynomials in \texttt{pols},
- \texttt{idx}: index of the series parameter, by default equals 1,
- \texttt{maxdeg}: maximal degree of the series,
- \texttt{nbr}: number of steps with Newton’s method,
**Verbose:** whether to write intermediate output to screen or not.

On return is a list of lists of strings. Each lists of strings represents the series solution for the variables in the list `pols`.

```python
series.quaddobl_pade_approximants(pols, sols, idx=1, numdeg=2, dendeg=2, nbr=4, verbose=True)
```

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,
- `verbose`: whether to write intermediate output to screen or not.

On return is a list of lists of strings. Each lists of strings represents the series solution for the variables in the list `pols`.

```python
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.

```python
series.replace_symbol(pol, idx)
```

In the polynomial `pol`, replaces the first symbol by the symbol at place `idx`.

```python
series.standard_newton_power_series(pols, lser, idx=1, maxdeg=4, nbr=4, checkin=True, verbose=True)
```

Computes series in standard 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.
- `verbose`: whether to write intermediate output to screen or not.

On return is a list of lists of strings. Each lists of strings represents the series solution for the variables in the list `pols`.

```python
series.standard_newton_series(pols, sols, idx=1, maxdeg=4, nbr=4, verbose=True)
```

Computes series in standard 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,
verbose: whether to write intermediate output to screen or not.

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.standard_pade_approximants(pols, sols, idx=1, numdeg=2, dendeg=2, nbr=4, verbose=True)
Computes Pade approximants based on the series in standard 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,
verbose: whether to write intermediate output to screen or not.

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.substitute_symbol(pols, idx)
Given in pols is a list of polynomials, replaces the first symbol by the symbol at place idx.

series.test(precision='d')
Tests the application of Newton’s method to compute power series solutions of a polynomial system.

series.viviani(prc='d')
Returns the system which stores the Viviani curve, with some solutions intersected with a plane, in double (‘d’), double double (‘dd’), or quad double (‘qd’) precision.

series.viviani2(precision='d')
Computes the power series expansion for the Viviani curve, from a natural parameter perspective. The default precision is double (‘d’). Other precisions are double double (‘dd’) and quad double (‘qd’).

4.6.4 functions in the module curves

Power series are input to Padé approximants, which provide accurate predictors to approximate algebraic curves.

The module curves exports functions to approximate algebraic space curves with rational expressions, in particular Padé approximants.

curves.dobldobl_closest_pole()
Returns a tuple with the real and imaginary part of the closest pole used in the predictor in double double precision. The result is only meaningful if the real part is positive. The double coefficients are the high parts of the double doubles.

curves.dobldobl_estimated_distance()
Returns the current estimated distance to the closest solution computed by the tracker in double double precision.

curves.dobldobl_get_solution(verbos=False)
Returns the current solution on the path, in double double precision, which starts at the solution set with dobldobl_set_solution(). If verbose, then extra output is written.
**curves.dobldobl_hessian_step()**

Returns the current Hessian step in the tracker in double double precision.

**curves.dobldobl_next_loop (hom, idx, sols, verbose=False)**

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. The function is interactive, prompting the user each time before performing the next predictor-corrector step. If verbose, then extra output is written. On return are the string representations of the solutions computed at the end of the paths.

**curves.dobldobl_next_track (target, start, sols, verbose=False, homogeneous=False)**

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. The function is interactive, prompting the user each time before performing the next predictor-corrector step. If verbose, 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.dobldobl_pade_coefficients (idx)**

Returns a tuple of lists with the coefficients of the Pade approximants computed by the predictor in double double precision. The double coefficients are the high parts of the double doubles. 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.dobldobl_pade_vector (dim)**

Returns the list of all coefficients over all *dim* variables, computed by the predictor in double double precision. The double coefficients are the high parts of the double doubles.

**curves.dobldobl_pole_radius ()**

Returns the smallest pole radius, used in the predictor in double double precision. The double on return is the high part of a double double.

**curves.dobldobl_pole_step ()**

Returns the current pole step in the tracker in double double precision.

**curves.dobldobl_poles (dim)**

Returns a list of lists of all poles of the vector of length *dim*, computed by the predictor in double double precision. The doubles in the poles are the high parts of the double doubles.

**curves.dobldobl_predict_correct (verbose=False)**

Performs one predictor and one corrector step on the set homotopy and the set solution, in double double precision. If verbose, then extra output is written.

**curves.dobldobl_series_coefficients (dim)**

Returns a list of lists with the coefficients of the series computed by the predictor in double double precision. The double coefficients are the high parts of the double doubles. On entry in *dim* is the number of variables.

**curves.dobldobl_series_step ()**

Returns the current series step in the tracker in double double precision.

**curves.dobldobl_set_homotopy (target, start, verbose=False, homogeneous=False)**

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 verbose, then extra output is written. If homogeneous, then path tracking happens in projective space, otherwise the original affine coordinates are used. Returns the failure code of the homotopy initializer.

---

**4.6. Newton polytopes, monomial maps, and power series**

143
curves.dobldobl_set_parameter_homotopy (hom, idx, verbose=False)
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 verbose, then extra output is written. Returns the failure code of the homotopy initializer.

curves.dobldobl_set_solution (nvar, sol, verbose=False)
Sets the start solution in sol for the step-by-step run of the series-Pade tracker, in double double precision. If verbose, then extra output is written. The number of variables is in nvar.

curves.dobldobl_step_size()
Returns the current step size in the tracker in double double precision.

curves.dobldobl_t_value()
Returns the current t value in the tracker in double double precision.

curves.dobldobl_track (target, start, sols, filename='', verbose=False, mhom=0, partition=None)
Wraps the tracker for Pade continuation in double double precision. On input are a target system, a start system with solutions, optionally: a string filename and the verbose flag. 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. By default mhom is zero and tracking 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 are the string representations of the solutions computed at the end of the paths.

curves.get_corrector_residual_tolerance()
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()
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()
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()
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_gamma_constant()
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_homotopy_continuation_parameter (idx)
Returns the current value of the homotopy continuation parameter with index idx, where idx is an integer in range(1, 13).

curves.get_maximum_corrector_steps()
Returns the current value of the maximum number of corrector steps executed after the predictor stage.

curves.get_maximum_step_size()
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()
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()
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_pole_radius_beta_factor()
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()
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_series_beta_factor()
Returns the current multiplication factor of the step size set by the power series approximation for the algebraic
curve.

curves.get_zero_series_coefficient_tolerance()
Returns the current tolerance on the series coefficient to be zero. A coefficient in a power series will be consid-
ered as zero if its absolute value drops below this tolerance.

curves.quaddobl_closest_pole()
Returns a tuple with the real and imaginary part of the closest pole used in the predictor in quad double precision.
The result is only meaningful is the real part is positive. The double coefficients are the highest parts of the quad
doubles.

curves.quaddobl_estimated_distance()
Returns the current estimated distance to the closest solution computed by the tracker in quad double precision.

curves.quaddobl_get_solution(verbos=False)
Returns the current solution on the path, in quad double precision, which starts at the solution set with quad-
dobl_set_solution(). If verbose, then extra output is written.

curves.quaddobl_hessian_step()
Returns the current Hessian step in the tracker in quad double precision.

curves.quaddobl_next_loop(hom, idx, sols, verbose=False)
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. The function is interactive, prompting the user each time before performing the
next predictor-corrector step. If verbose, then extra output is written. On return are the string representations
of the solutions computed at the end of the paths.

curves.quaddobl_next_track(target, start, sols, verbose=False, homogeneous=False)
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 poly-
nomials 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. The
function is interactive, prompting the user each time before performing the next predictor-corrector step. If
verbose, 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.quaddobl_pade_coefficients(idx)
Returns a tuple of lists with the coefficients of the Pade approximants computed by the predictor in quad double precision. The double coefficients are the highest parts of the quad doubles. 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.quaddobl_pade_vector(dim)
Returns the list of all coefficients over all dim variables, computed by the predictor in quad double precision. The double coefficients are the highest parts of the quad doubles.

curves.quaddobl_pole_radius()
Returns the smallest pole radius, used in the predictor in quad double precision. The double on return is the highest part of a quad double.

curves.quaddobl_pole_step()
Returns the current pole step in the tracker in quad double precision.

curves.quaddobl_poles(dim)
Returns a list of lists of all poles of the vector of length dim, computed by the predictor in quad double precision. The doubles in the poles are the highest parts of the quad doubles.

curves.quaddobl_predict_correct(verbos=False)
Performs one predictor and one corrector step on the set homotopy and the set solution, in quad double precision. If verbose, then extra output is written.

curves.quaddobl_series_coefficients(dim)
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.quaddobl_series_step()
Returns the current series step in the tracker in quad double precision.

curves.quaddobl_set_homotopy(target, start, verbose=False, homogeneous=False)
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 verbose, then extra output is written. If homogeneous, then path tracking happens in projective space, otherwise the original affine coordinates are used. Returns the failure code of the homotopy initializer.

curves.quaddobl_set_parameter_homotopy(hom, idx, verbose=False)
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 verbose, then extra output is written. Returns the failure code of the homotopy initializer.

curves.quaddobl_set_solution(nvar, sol, verbose=False)
Sets the start solution in sol for the step-by-step run of the series-Pade tracker, in quad double precision. If verbose, then extra output is written. The number of variables is in nvar.

curves.quaddobl_step_size()
Returns the current step size in the tracker in quad double precision.

curves.quaddobl_t_value()
Returns the current t value in the tracker in quad double precision.

curves.quaddobl_track(target, start, sols, filename='', verbose=False, mhom=0, partition=None)
Wraps the tracker for Pade continuation in quad double precision. On input are a target system, a start system with solutions, optionally: a string filename and the verbose flag. 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. By default mhom is zero and tracking happens in the original coordinates, if mhom equals one, then

Chapter 4. Reference Manual
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 are the string representations of the solutions computed at the end of the paths.

\[ \text{curves.set_homotopy_continuation_gamma}(regamma=0, imgamma=0) \]

Sets the value of the homotopy continuation gamma constant to the complex number with real part in \( regamma \) and the imaginary part in \( imgamma \). If both \( regamma \) and \( imgamma \) are zero, then the user is prompted to provide values for \( regamma \) and \( imgamma \).

\[ \text{curves.set_homotopy_continuation_parameter}(idx, val) \]

Sets the value of the homotopy continuation parameter with index \( idx \), where \( idx \) is in an integer in range(2, 13), to the value \( val \).

\[ \text{curves.standard_closest_pole}() \]

Returns a tuple with the real and imaginary part of the closest pole used in the predictor in standard double precision. The result is only meaningful is the real part is positive.

\[ \text{curves.standard_estimated_distance}() \]

Returns the current estimated distance to the closest solution computed by the tracker in double precision.

\[ \text{curves.standard_get_solution}(verbose=False) \]

Returns the current solution on the path, in double precision, which starts at the solution set with \( \text{standard_set_solution}() \). If \( verbose \), then extra output is written.

\[ \text{curves.standard_hessian_step}() \]

Returns the current Hessian step in the tracker in double precision.

\[ \text{curves.standard_next_loop}(hom, idx, sols, verbose=False) \]

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. The function is interactive, prompting the user each time before performing the next predictor-corrector step. If \( verbose \), then extra output is written. On return are the string representations of the solutions computed at the end of the paths.

\[ \text{curves.standard_next_track}(target, start, sols, verbose=False, homogeneous=False) \]

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. The function is interactive, prompting the user each time before performing the next predictor-corrector step. If \( verbose \), 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.

\[ \text{curves.standard_pade_coefficients}(idx) \]

Returns a tuple of lists with the coefficients of the Pade approximants computed by the predictor in standard 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.

\[ \text{curves.standard_pade_vector}(dim) \]

Returns the list of all coefficients over all \( dim \) variables, computed by the predictor in standard double precision.

\[ \text{curves.standard_pole_radius}() \]

Returns the smallest pole radius, used in the predictor in standard double precision.

\[ \text{curves.standard_pole_step}() \]

Returns the current pole step in the tracker in double precision.
curves.standard_poles(dim)
    Returns a list of lists of all poles of the vector of length dim, computed by the predictor in standard double precision.

curves.standard_predict_correct(verbos=False)
    Performs one predictor and one corrector step on the set homotopy and the set solution, in standard double precision. If verbose, then extra output is written.

curves.standard_series_coefficients(dim)
    Returns a list of lists with the coefficients of the series computed by the predictor in standard double precision. On entry in dim is the number of variables.

curves.standard_series_step()
    Returns the current series step in the tracker in double precision.

curves.standard_set_homotopy(target, start, verbose=False, homogeneous=False)
    Initializes the homotopy with the target and start system for a step-by-step run of the series-Pade tracker, in double precision. If verbose, then extra output is written. If homogeneous, then path tracking happens in projective space, otherwise the original affine coordinates are used. Returns the failure code of the homotopy initializer.

curves.standard_set_parameter_homotopy(hom, idx, verbose=False)
    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 verbose, then extra output is written. Returns the failure code of the homotopy initializer.

curves.standard_set_solution(nvar, sol, verbose=False)
    Sets the start solution in sol for the step-by-step run of the series-Pade tracker, in standard double precision. If verbose, then extra output is written. The number of variables is in nvar.

curves.standard_step_size()
    Returns the current step size in the tracker in double precision.

curves.standard_t_value()
    Returns the current t value in the tracker in double precision.

curves.standard_track(target, start, sols, filename='', verbose=False, mhom=0, partition=None)
    Wraps the tracker for Pade continuation in standard double precision. On input are a target system, a start system with solutions, optionally: a string filename and the verbose flag. 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. By default mhom is zero and tracking 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 are the string representations of the solutions computed at the end of the paths.

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_hyperbola(precision='d')
    Tests the step-by-step Pade tracker on a hyperbola, by default in double precision (when precision equals ‘d’). Other supported precisions are double double (‘dd’) and quad double (‘qd”).
curves.test_next_track (precision='d')
Tunes the parameters and runs the step-by-step tracker. The precision is either double ('d'), double double ('dd'), or quad double ('qd').

curves.test_simple_track (precision='d')
Tunes the parameters and runs a simple test on the trackers. The precision is either double ('d'), double double ('dd'), or quad double ('qd').

curves.test_twohom (precision='d')
Test on a 2-homogeneous homotopy.

curves.tune_homotopy_continuation_parameters()
The interactive loop prompts the user to tune the parameters.

curves.write_homotopy_continuation_parameters()
Writes the values of the homotopy continuation parameters.

4.7 a graphical user interface

With Tkinter we can develop a graphical user interface. The module exports some possible development for a GUI to solve polynomial systems and to process solutions.

4.7.1 functions in the module dashboard

This module prototypes a graphical user interface to phcpy.

class dashboard.CoordinatePlot (wdw, dim, sols, idx)
Shows the distribution of one coordinate of a list of solutions in the complex plane.

    plot()
    Plots a coordinate of the list of solutions.

class dashboard.Scroller (wdw, sols)
Scrolls through a solution list.

    next()
    Increases the cursor by one if possible.

    previous()
    Decreases the cursor by one if possible.

    show()
    Shows the solution at position self.cursor in the message widget and updates the entry widget.

class dashboard.SolveButton (wdw, pols=[])
Simple input/output text widget and button graphical user interface to a solver.

    solve()
    Takes the data from the input text widget, places the data into a list of polynomials, calls the blackbox solver and displays the solutions in the output text widget.

dashboard.launchsolver (pols=[])
Instantiates a Tk object and launches the event loop.

dashboard.plotcoordinate (sols, idx)
Instantiates CoordinatePlot with a Tk object and launches the main event loop.

dashboard.pols2str (pols)
Returns the input string to put into the text input widget for the string representations of polynomials in pols.
dashboard.**scrollsols** *(sols)*
Instantiates the window and launches the GUI to scroll through the solutions in the list sols.

dashboard.**str2pols** *(strp)*
Returns the list of string representations of the polynomials in the string strp.

dashboard.**testcoordinateplot**()
Solves the cyclic 5-roots problem, prompts the user for an index of a coordinate, and launches the plotcoordinate function.

dashboard.**testscroller**()
Solves the cyclic 5-roots problems and launches the scroller.

dashboard.**testsolvebutton**()
Solves the cyclic 5-roots problem and launches the solve button.

dashboard.**windowsize** *(sols, idx)*
Returns the minimal and maximal value of the real and imaginary parts of the coordinate with index idx of the list of solutions in sols, as a tuple of 4 values: (realmin, realmax, imagmin, imagmax).

### 4.8 the module phcpy.phcpy2c3

Almost all computations in phcpy are done with compiled code, provided in one object.

#### 4.8.1 functions in the module interface

Polynomial systems and solutions are passed through as strings. The coefficients of the polynomials and coordinates of the solutions are evaluated in standard double, double double, quad double precision, or arbitrary multiprecision.

This module provides the data interface to PHCpack. The user gives as input string representations of polynomials or solutions to the interface functions which store the data.

- **interface.load_dobldobl_laurent_system**()
  Returns the Laurent polynomials stored in the system container with double double complex coefficients.

- **interface.load_dobldobl_solutions**()
  Returns the list of solutions stored in the container for complex double double solutions.

- **interface.load_dobldobl_system**()
  Returns the polynomials stored in the system container with double double complex coefficients.

- **interface.load_multprec_laurent_system**()
  Returns the Laurent polynomials stored in the system container with multiprecision complex coefficients.

- **interface.load_multprec_solutions**()
  Returns the list of solutions stored in the container for complex multiprecision solutions.

- **interface.load_multprec_system**()
  Returns the polynomials stored in the system container with arbitrary multiprecision complex coefficients.

- **interface.load_quaddobl_laurent_system**()
  Returns the Laurent polynomials stored in the system container with quad double complex coefficients.

- **interface.load_quaddobl_solutions**()
  Returns the list of solutions stored in the container for complex quad double solutions.

- **interface.load_quaddobl_system**()
  Returns the polynomials stored in the system container with quad double complex coefficients.
interface.load_standard_laurent_system()
    Returns the Laurent polynomials stored in the system container for standard double precision arithmetic.

interface.load_standard_solutions()
    Returns the list of solutions stored in the container for solutions with standard double precision.

interface.load_standard_system()
    Returns the polynomials stored in the system container for standard double precision arithmetic.

interface.load_standard_tableau(verbos=False)
    Returns the tableau form of the system stored in the container for double precision coefficients.

interface.read_dobldobl_solutions(filename)
    Returns the list of solutions stored on file with the given file name. The solutions are parsed in double double precision.

interface.read_dobldobl_system(filename)
    Opens the filename for reading a polynomial system with coefficients in double double precision. Returns the list of polynomials in the system or None if something went wrong.

interface.read_dobldobl_system_and_solutions(filename)
    Opens the filename for reading a polynomial system with coefficients in double double precision, and its corresponding list of solutions. Returns None if the reading went wrong, or otherwise returns a tuple with first the list of polynomials and second the list of solutions.

interface.read_quaddobl_solutions(filename)
    Returns the list of solutions stored on file with the given file name. The solutions are parsed in quad double precision.

interface.read_quaddobl_system(filename)
    Opens the filename for reading a polynomial system with coefficients in quad double precision. Returns the list of polynomials in the system or None if something went wrong.

interface.read_quaddobl_system_and_solutions(filename)
    Opens the filename for reading a polynomial system with coefficients in quad double precision, and its corresponding list of solutions. Returns None if the reading went wrong, or otherwise returns a tuple with first the list of polynomials and second the list of solutions.

interface.read_standard_solutions(filename)
    Returns the list of solutions stored on file with the given file name. The solutions are parsed in standard double precision.

interface.read_standard_system(filename)
    Opens the filename for reading a polynomial system with coefficients in standard double precision. Returns the list of polynomials in the system or None if something went wrong.

interface.read_standard_system_and_solutions(filename)
    Opens the filename for reading a polynomial system with coefficients in standard double precision, and its corresponding list of solutions. Returns None if the reading went wrong, or otherwise returns a tuple with first the list of polynomials and second the list of solutions.

interface.store_dobldobl_laurent_system(polsys, **nbvar)
    Stores the Laurent polynomials represented by the list of strings in polsys into the container for systems with coefficients in double double precision. If nbvar is omitted, then the system is assumed to be square. Otherwise, suppose the number of variables equals 2 and pols is the list of polynomials, then store_dobldobl_laurent_system(pols, nbvar=2) stores the polynomials in pols in the dobldobl Laurent systems container.

interface.store_dobldobl_laurent_witness_set(nbvar, dim, pols, sols)
    Given in nbvar 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 \( \text{dim} \) equals the number of slack variables used in the embedding of \( \text{pols} \) and \( \text{sols} \). This \( \text{dim} \) also equals the dimension of the solution set represented by the witness set given by the lists \( \text{pols} \) and \( \text{sols} \). The symbols for the slack variables are swapped to the end of the symbol table in both the Laurent polynomials and the solutions.

**interface.\text{store\_dobldobl\_solutions}(nvar, sols)**

Stores the solutions in the list \( \text{sols} \), represented as strings in PHCpack format into the solution container for processing with complex double double arithmetic. The number \( nvar \) equals the number of variables.

**interface.\text{store\_dobldobl\_system}(\text{polsys}, **nbvar)**

Stores the polynomials represented by the list of strings in \( \text{polsys} \) into the systems container for double double arithmetic. The number of variables is an optional argument given in \( \text{nbvar} \). If \( \text{nbvar} \) is omitted, then the system is assumed to be square. Otherwise, suppose the number of variables equals 2 and \( \text{pols} \) is the list of polynomials, then the call \text{store\_dobldobl\_system}(\text{pols}, \text{nbvar}=2) will store the polynomials in \( \text{pols} \) in the dobldobl systems container.

**interface.\text{store\_dobldobl\_witness\_set}(nbvar, dim, pols, sols)**

Given in \( \text{nbvar} \) is the total number of variables in the list of polynomials in \( \text{pols} \) and its list of solutions in \( \text{sols} \). The coefficients in the polynomials and the coordinates of the solutions will be parsed and stored in double double precision. The parameter \( \text{dim} \) equals the number of double variables used in the embedding of \( \text{pols} \) and \( \text{sols} \). This \( \text{dim} \) also equals the dimension of the solution set represented by the witness set given by the lists \( \text{pols} \) and \( \text{sols} \). The symbols for the slack variables are swapped to the end of the symbol table in both the polynomials and the solutions.

**interface.\text{store\_multprec\_laurent\_system}(\text{polsys}, \text{decimals}, **nbvar)**

Stores the Laurent polynomials represented by the list of strings in \( \text{polsys} \) into the container for systems with coefficients in multiprecision. The parameter \( \text{decimals} \) equals the number of decimal places in the working precision for the parsing of the strings in \( \text{polsys} \). If \( \text{nbvar} \) is omitted, then the system is assumed to be square. Otherwise, suppose the number of variables equals 2 and \( \text{pols} \) is the list of polynomials, then \text{store\_multprec\_laurent\_system}(\text{pols}, \text{nbvar}=2) stores the polynomials in \( \text{pols} \) in the multprec Laurent systems container.

**interface.\text{store\_multprec\_solutions}(nvar, sols)**

Stores the solutions in the list \( \text{sols} \), represented as strings in PHCpack format into the solution container for processing with complex multiprecision arithmetic. The number \( nvar \) equals the number of variables.

**interface.\text{store\_multprec\_system}(\text{polsys}, \text{decimals}, **nbvar)**

Stores the polynomials represented by the list of strings in \( \text{polsys} \) into the systems container for multiprecision arithmetic. The parameter \( \text{decimals} \) equals the number of decimal places in the working precision for the parsing of the strings in \( \text{polsys} \). The number of variables is an optional argument given in \( \text{nbvar} \). If \( \text{nbvar} \) is omitted, then the system is assumed to be square. Otherwise, suppose the number of variables equals 2 and \( \text{pols} \) is the list of polynomials, then the call \text{store\_multprec\_system}(\text{pols}, \text{nbvar}=2) will store the polynomials in \( \text{pols} \) in the multiprecision systems container.

**interface.\text{store\_quaddobl\_laurent\_system}(\text{polsys}, **nbvar)**

Stores the Laurent polynomials represented by the list of strings in \( \text{polsys} \) into the container for systems with coefficients in quad double precision. If \( \text{nbvar} \) is omitted, then the system is assumed to be square. Otherwise, suppose the number of variables equals 2 and \( \text{pols} \) is the list of polynomials, then \text{store\_quaddobl\_laurent\_system}(\text{pols}, \text{nbvar}=2) stores the polynomials in \( \text{pols} \) in the quaddobl Laurent systems container.

**interface.\text{store\_quaddobl\_laurent\_witness\_set}(\text{nbvar}, \text{dim}, \text{pols}, \text{sols})**

Given in \( \text{nbvar} \) is the total number of variables in the list of Laurent polynomials in \( \text{pols} \) and its list of solutions in \( \text{sols} \). The coefficients in the Laurent polynomials and the coordinates of the solutions will be parsed and stored in quad double precision. The parameter \( \text{dim} \) equals the number of slack variables used in the embedding of \( \text{pols} \) and \( \text{sols} \). This \( \text{dim} \) also equals the dimension of the solution set represented by the witness set given by the lists \( \text{pols} \) and \( \text{sols} \). The symbols for the slack variables are swapped to the end of the symbol table in both the Laurent polynomials and the solutions.
interface.\textbf{store\_quaddobl\_solutions}(nvar, sols)
Stores the solutions in the list \textit{sols}, represented as strings in PHCpack format into the solution container for processing with complex quad double arithmetic. The number \textit{nvar} equals the number of variables.

interface.\textbf{store\_quaddobl\_system}(polsys, **nbvar)
Stores the polynomials represented by the list of strings in \textit{polsys} into the systems container for quad double arithmetic. The number of variables is an optional argument given in \textit{nbvar}. If \textit{nbvar} is omitted, then the system is assumed to be square. Otherwise, suppose the number of variables equals 2 and \textit{pols} is the list of polynomials, then the call \textbf{store\_quaddobl\_system}(\textit{pols}, nbvar=2) will store the polynomials in \textit{pols} in the quaddobl systems container.

interface.\textbf{store\_quaddobl\_witness\_set}(nbvar, dim, pols, sols)
Given in \textit{nbvar} is the total number of variables in the list of polynomials in \textit{pols} and its list of solutions in \textit{sols}. The coefficients in the polynomials and the coordinates of the solutions will be parsed and stored in quad double precision. The parameter \textit{dim} equals the number of slack variables used in the embedding of \textit{pols} and \textit{sols}. This \textit{dim} also equals the dimension of the solution set represented by the witness set given by the lists \textit{pols} and \textit{sols}. The symbols for the slack variables are swapped to the end of the symbol table in both the polynomials and the solutions.

interface.\textbf{store\_standard\_laurent\_system}(polsys, **nbvar)
Stores the Laurent polynomials represented by the list of strings in \textit{polsys} into the container for systems with coefficients in standard double precision. If \textit{nbvar} is omitted, then the system is assumed to be square. Otherwise, suppose the number of variables equals 2 and \textit{pols} is the list of polynomials, then \textbf{store\_standard\_laurent\_system}(\textit{pols}, nbvar=2) stores the polynomials in \textit{pols} in the standard Laurent systems container.

interface.\textbf{store\_standard\_witness\_set}(nbvar, dim, pols, sols)
Given in \textit{nbvar} is the total number of variables in the list of Laurent polynomials in \textit{pols} and its list of solutions in \textit{sols}. The coefficients in the Laurent polynomials and the coordinates of the solutions will be parsed and stored in standard double precision. The parameter \textit{dim} equals the number of slack variables used in the embedding of \textit{pols} and \textit{sols}. This \textit{dim} also equals the dimension of the solution set represented by the witness set given by the lists \textit{pols} and \textit{sols}. The symbols for the slack variables are swapped to the end of the symbol table in both the Laurent polynomials and the solutions.

interface.\textbf{store\_standard\_solutions}(nvar, sols)
Stores the solutions in the list \textit{sols}, represented as strings in PHCpack format into the container for solutions with standard double precision. The number \textit{nvar} equals the number of variables.

interface.\textbf{store\_standard\_system}(polsys, **nbvar)
Stores the polynomials represented by the list of strings in \textit{polsys} into the container for systems with coefficients in standard double precision. The number of variables is an optional argument given in \textit{nbvar}. If \textit{nbvar} is omitted, then the system is assumed to be square. Otherwise, suppose the number of variables equals 2 and \textit{pols} is the list of polynomials, then the call \textbf{store\_standard\_system}(\textit{pols}, nbvar=2) will store the polynomials in \textit{pols} in the standard systems container.

interface.\textbf{store\_standard\_tableau}(poltab, verbose=False)
Stores the polynomial system given in the list of lists \textit{poltab} in the container for systems with coefficients in standard double precision. Every polynomial in the system is represented by a list of tuples. A monomial is represented by a 2-tuple:
1. the coefficient of the monomial is a complex number,
2. the exponent are a tuple of natural numbers.

For example, the system $x^2 - y = 0, x^3 - z = 0$ is represented as $\{[(1+0j), (2, 0, 0)), ((1+0j), (0, 1, 0))], \{[(1+0j), (3, 0, 0)), ((1+0j), (0, 0, 1))\}]$.

interface.\textbf{store\_standard\_witness\_set}(nbvar, dim, pols, sols)
Given in \textit{nbvar} is the total number of variables in the list of polynomials in \textit{pols} and its list of solutions in \textit{sols}.
The coefficients in the polynomials and the coordinates of the solutions will be parsed and stored in standard double precision. The parameter \textit{dim} equals the number of slack variables used in the embedding of \textit{pols} and \textit{sols}. This \textit{dim} also equals the dimension of the solution set represented by the witness set given by the lists \textit{pols} and \textit{sols}. The symbols for the slack variables are swapped to the end of the symbol table in both the polynomials and the solutions.

\begin{verbatim}
interface.test (prc='d', laurent=False)
Tests the storing of a witness set for the twisted cubic. The embedding induces the order x, y, zz1, z on the variables. After storing the witness set, the order is x, y, z, zz1, in both the system and solutions. The default precision prc is double ‘d’. Other supported precisions are double double ‘dd’ and quad double ‘qd’.
\end{verbatim}

4.8.2 functions in the module phcpy2c3

The module \texttt{phcpy2c3} wraps the C functions in the C interface to PHCpack. The C interface to PHCpack was developed in the application of message passing (MPI) to run the path trackers on distributed memory multiprocessing computers. All functions documented below have their counterpart in C that are therefore then also directly accessible from C programs.

\begin{verbatim}
phcpy2c3.py2c_PHCpack_version_string()
Returns the version string of PHCpack. The version string is 40 characters long.
\end{verbatim}

\begin{verbatim}
phcpy2c3.py2c_ade_manypaths_d()
Tracks many solution paths with algorithmic differentiation in double precision on the data in the systems and solutions container. The start and target systems must have been defined and the standard solutions container holds valid solutions. On entry is the verbose flag, which equals zero if no output is wanted, or 1 if extra information should be written to screen. On return is the failure code, which equals zero if all went well.
\end{verbatim}

\begin{verbatim}
phcpy2c3.py2c_ade_manypaths_d_pars()
Tracks many solution paths with algorithmic differentiation in double precision on the data in the systems and solutions container. The start and target systems must have been defined and the standard solutions container holds valid solutions. On entry is the verbose flag, which equals zero if no output is wanted, or 1 if extra information should be written to screen. Other input parameters are the real and imaginary parts of the gamma constant. Then, the 14 values of the path parameters has to be provided. On return is the failure code, which equals zero if all went well.
\end{verbatim}

\begin{verbatim}
phcpy2c3.py2c_ade_manypaths_dd()
Tracks many solution paths with algorithmic differentiation in double precision on the data in the systems and solutions container. The start and target systems must have been defined and the dobldobl solutions container holds valid solutions. On entry is the verbose flag, which equals zero if no output is wanted, or 1 if extra information should be written to screen. On return is the failure code, which equals zero if all went well.
\end{verbatim}

\begin{verbatim}
phcpy2c3.py2c_ade_manypaths_dd_pars()
Tracks many solution paths with algorithmic differentiation in double double precision on the data in the systems and solutions container. The start and target systems must have been defined and the dobldobl solutions container holds valid solutions. On entry is the verbose flag, which equals zero if no output is wanted, or 1 if extra information should be written to screen. Other input parameters are the real and imaginary parts of the gamma constant. Then, the 14 values of the path parameters has to be provided. On return is the failure code, which equals zero if all went well.
\end{verbatim}

\begin{verbatim}
phcpy2c3.py2c_ade_manypaths_qd()
Tracks many solution paths with algorithmic differentiation in quad double precision on the data in the systems and solutions container. The start and target systems must have been defined and the quaddobl solutions container holds valid solutions. On entry is the verbose flag, which equals zero if no output is wanted, or 1 if extra information should be written to screen. On return is the failure code, which equals zero if all went well.
\end{verbatim}
Tracks many solution paths with algorithmic differentiation in quad double precision on the data in the systems and solutions container. The start and target systems must have been defined and the quaddobl solutions container holds valid solutions. On entry is the verbose flag, which equals zero if no output is wanted, or 1 if extra information should be written to screen. Other input parameters are the real and imaginary parts of the gamma constant. Then, the 14 values of the path parameters has to be provided. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_ade_newton_d()**

Runs Newton’s method with algorithmic differentiation in double precision on the data in the systems and solutions container. The standard systems container must contain a valid polynomial system and the standard solutions container must hold a valid solution. On entry is the verbose flag, which equals zero if no output is wanted, or 1 if extra information should be written to screen. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_ade_newton_dd()**

Runs Newton’s method with algorithmic differentiation in double double precision on the data in the systems and solutions container. The dobldobl systems container must contain a valid polynomial system and the dobldobl solutions container must hold a valid solution. On entry is the verbose flag, which equals zero if no output is wanted, or 1 if extra information should be written to screen. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_ade_newton_qd()**

Runs Newton’s method with algorithmic differentiation in quad double precision on the data in the systems and solutions container. The quaddobl systems container must contain a valid polynomial system and the quaddobl solutions container must hold a valid solution. On entry is the verbose flag, which equals zero if no output is wanted, or 1 if extra information should be written to screen. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_ade_onepath_d()**

Tracks one solution path with algorithmic differentiation in double precision on the data in the systems and solutions container. The start and target systems must have been defined and the standard solutions container must holds valid solution. On entry is the verbose flag, which equals zero if no output is wanted, or 1 if extra information should be written to screen. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_ade_onepath_dd()**

Tracks one solution path with algorithmic differentiation in double double precision on the data in the systems and solutions container. The start and target systems must have been defined and the dobldobl solutions container must holds valid solution. On entry is the verbose flag, which equals zero if no output is wanted, or 1 if extra information should be written to screen. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_ade_onepath_qd()**

Tracks one solution path with algorithmic differentiation in quad double precision on the data in the systems and solutions container. The start and target systems must have been defined and the quaddobl solutions container must holds valid solution. On entry is the verbose flag, which equals zero if no output is wanted, or 1 if extra information should be written to screen. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_autotune_continuation_parameters()**

Tunes the values of the continuation parameters. On input are two integers: 1) the difficulty level of the solution paths; and 2) the number of decimal places in the precision.

**phcpy2c3.py2c_celcon_append_lifted_point()**

Appends a lifted point to the cells container. There are three input parameters: 1) the dimension of the point; 2) the index of the support to where to append to; and 3) the string representation of the lifted point. Returns the failure code, which equals zero when all went well.

**phcpy2c3.py2c_celcon_clear_container()**

Deallocates the data in the cell container.
The random coefficient system in double double precision is copied from the cell container to the container for systems with coefficients in double double precision.

The random coefficient system in quad double precision is copied from the cell container to the container for systems with coefficients in quad double precision.

The random coefficient system in standard double precision is copied from the cell container to the container for systems with coefficients in standard double precision.

Copies the i-th target solution corresponding to the k-th mixed cell to the container for solutions in double double precision. There are two input parameters for this function: 1) k, the index to the mixed cell; 2) i, the index to the i-th solution path defined by the cell. On return is the failure code, which equals zero when all went well.

Copies the i-th target solution corresponding to the k-th mixed cell to the container for solutions in quad double precision. There are two input parameters for this function: 1) k, the index to the mixed cell; 2) i, the index to the i-th solution path defined by the cell. On return is the failure code, which equals zero when all went well.

Copies the i-th target solution corresponding to the k-th mixed cell to the container for solutions in standard double precision. There are two input parameters for this function: 1) k, the index to the mixed cell; 2) i, the index to the i-th solution path defined by the cell. On return is the failure code, which equals zero when all went well.

Based on the lifting and the random coefficient system, the polyhedral homotopy to solve the random coefficient system in double double precision is constructed. This function also initializes the internal data structures to store the solutions of start and target systems. The lifted supports and the random coefficient system are defined. On return is the failure code, which equals zero if all went well.

Based on the lifted supports stored in the container, a random coefficient system with coefficients in double double precision is stored in the cell container.

Initializes the cell container with the number of distinct supports, this number is given as the one input parameter. On return is the failure code, which equals zero if all went well.

returns 1 if stable mixed cells were stored, 0 otherwise

Returns the mixed volume of the supports stored in the cell container.

returns the number of cells in the cell container

returns the number of original cells in the cell container

returns the number of stable cells in the cell container

Permutes the systems in the container for polynomial and Laurent systems with double double coefficients corresponding to the permutation used to compute the mixed-cell configuration. On return is the failure code, which equals zero if all went well.
Permutes the systems in the container for polynomial and Laurent systems with quad double coefficients corresponding to the permutation used to compute the mixed-cell configuration. On return is the failure code, which equals zero if all went well.

Permutes the systems in the container for polynomial and Laurent systems with standard double coefficients corresponding to the permutation used to compute the mixed-cell configuration. On return is the failure code, which equals zero if all went well.

Based on the lifting and the random coefficient system, the polyhedral homotopy to solve the random coefficient system in quad double precision is constructed. This function also initializes the internal data structures to store the solutions of start and target systems. The lifted supports and the random coefficient system are defined. On return is the failure code, which equals zero if all went well.

Based on the lifted supports stored in the container, a random coefficient system with coefficients in quad double precision is stored in the cell container.

Returns a string representation of a lifted point. On input are three integer numbers: 1) the number of coordinates in the lifted point; 2) the index to the support set; and 3) the index to the point in that support set.

Defines the type of mixture of the support sets. On input are two parameters, an integer and a string: 1) the integer equals the number of distinct supports; 2) the string is a string representation of a Python list of integers, there are as many integers as the value of the first parameter. Each integer is a positive number, equal to the number of occurrences of each support set.

Solves the start system corresponding to the k-th mixed cell, using double double precision arithmetic. The precondition for this function is that the creation of the polyhedral homotopy in double double precision ended well. On return is the number of solution found, which must equal the mixed volume of the k-th mixed cell.

Solves the start system corresponding to the k-th mixed cell, using quad double precision arithmetic. The precondition for this function is that the creation of the polyhedral homotopy in quad double precision ended well. On return is the number of solution found, which must equal the mixed volume of the k-th mixed cell.

Solves the start system corresponding to the k-th stable mixed cell, using double double precision arithmetic. The precondition for this function is that the creation of the polyhedral homotopy in double double precision ended well. On return is the number of solution found, which must equal the mixed volume of the k-th stable mixed cell.

Solves the start system corresponding to the k-th stable mixed cell, using quad double precision arithmetic. The precondition for this function is that the creation of the polyhedral homotopy in quad double precision ended well. On return is the number of solution found, which must equal the mixed volume of the k-th stable mixed cell.

Solves the start system corresponding to the k-th stable mixed cell, using standard double precision arithmetic. The precondition for this function is that the creation of the polyhedral homotopy in standard double precision ended well. On return is the number of solution found, which must equal the mixed volume of the k-th stable mixed cell.
Solves the start system corresponding to the k-th mixed cell, using standard double precision arithmetic. The precondition for this function is that the creation of the polyhedral homotopy in standard double precision ended well. On return is the number of solution found, which must equal the mixed volume of the k-th mixed cell.

phcpy2c3.py2c_celcon_standard_polyhedral_homotopy()
Based on the lifting and the random coefficient system, the polyhedral homotopy to solve the random coefficient system in standard double precision is constructed. This function also initializes the internal data structures to store the solutions of start and target systems. The lifted supports and the random coefficient system are defined. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_celcon_standard_random_coefficient_system()
Based on the lifted supports stored in the container, a random coefficient system with coefficients in standard double precision is stored in the cell container.

phcpy2c3.py2c_celcon_track_dobldobl_solution_path()
Tracks a solution path starting at the i-th solution of the k-th cell, using double double precision arithmetic. The precondition for this function is that the start system defined by the k-th mixed cell is solved in double double precision. There are three input parameters: 1) k, the index to a mixed cell in the cell container; 2) i, the index to a solution path defined by that mixed cell; 3) otp, the level for intermediate output during path tracking. A target solution corresponding to the k-th cell is added on return.

phcpy2c3.py2c_celcon_track_quaddobl_solution_path()
Tracks a solution path starting at the i-th solution of the k-th cell, using quad double precision arithmetic. The precondition for this function is that the start system defined by the k-th mixed cell is solved in quad double precision. There are three input parameters: 1) k, the index to a mixed cell in the cell container; 2) i, the index to a solution path defined by that mixed cell; 3) otp, the level for intermediate output during path tracking. A target solution corresponding to the k-th cell is added on return.

phcpy2c3.py2c_celcon_track_standard_solution_path()
Tracks a solution path starting at the i-th solution of the k-th cell, using standard double precision arithmetic. The precondition for this function is that the start system defined by the k-th mixed cell is solved in standard double precision. There are three input parameters: 1) k, the index to a mixed cell in the cell container; 2) i, the index to a solution path defined by that mixed cell; 3) otp, the level for intermediate output during path tracking. A target solution corresponding to the k-th cell is added on return.

phcpy2c3.py2c_celcon_type_of_mixture()
Returns the string representation of the type of mixture of the support sets. This string is the string representation of a Python list of integers.

phcpy2c3.py2c_clear_dobldobl_Laurent_data()
Deallocation of the homotopy stored in double double precision. On return is the failure code, which equals zero if all is well.

phcpy2c3.py2c_clear_dobldobl_operations_data()
Deallocation of the homotopy stored in double double precision. On return is the failure code, which equals zero if all is well.

phcpy2c3.py2c_clear_dobldobl_tracker()
Deallocation of the homotopy stored in double double precision. On return is the failure code, which equals zero if all is well.

phcpy2c3.py2c_clear_multprec_homotopy()
Deallocation of the homotopy stored in arbitrary multiprecision. On return is the failure code, which equals zero if all is well.
phcpy2c3.\texttt{py2c\_clear\_multprec\_tracker}()
    Deallocates data used in the arbitrary multiprecision tracker with a generator.

phcpy2c3.\texttt{py2c\_clear\_quaddobl\_Laurent\_data}()
    Deallocates data used to solve Laurent systems by homotopy continuation in quad double precision.

phcpy2c3.\texttt{py2c\_clear\_quaddobl\_homotopy}()
    Deallocation of the homotopy stored in quad double precision. On return is the failure code, which equals zero
    if all is well.

phcpy2c3.\texttt{py2c\_clear\_quaddobl\_operations\_data}()
    Deallocates the data used by solve\_by\_quaddobl\_homotopy\_continuation.

phcpy2c3.\texttt{py2c\_clear\_quaddobl\_tracker}()
    Deallocates data used in the quad double precision tracker with a generator.

phcpy2c3.\texttt{py2c\_clear\_quaddobl\_witsols}()
    Clears the witness solutions in quad double precision.

phcpy2c3.\texttt{py2c\_clear\_standard\_Laurent\_data}()
    Deallocates data used to solve Laurent systems by homotopy continuation in standard double precision.

phcpy2c3.\texttt{py2c\_clear\_standard\_homotopy}()
    Deallocation of the homotopy stored in standard double precision. On return is the failure code, which equals zero
    if all is well.

phcpy2c3.\texttt{py2c\_clear\_standard\_operations\_data}()
    Deallocates the data used by solve\_by\_standard\_homotopy\_continuation.

phcpy2c3.\texttt{py2c\_clear\_standard\_tracker}()
    Deallocates data used in the standard double precision tracker with a generator.

phcpy2c3.\texttt{py2c\_clear\_standard\_witsols}()
    Clears the witness solutions in standard double precision.

phcpy2c3.\texttt{py2c\_clear\_varbprec\_tracker}()
    Deallocates data used in the variable precision tracker with a generator.

phcpy2c3.\texttt{py2c\_copy\_dobldobl\_Laurent\_container\_to\_start\_system}()
    Copies the Laurent system in double double precision from the container to the start system.

phcpy2c3.\texttt{py2c\_copy\_dobldobl\_Laurent\_container\_to\_target\_system}()
    Copies the Laurent system in double double precision from the container to the target system.

phcpy2c3.\texttt{py2c\_copy\_dobldobl\_Laurent\_start\_system\_to\_container}()
    Copies the start Laurent system in double double precision to the systems container for Laurent systems.

phcpy2c3.\texttt{py2c\_copy\_dobldobl\_Laurent\_target\_system\_to\_container}()
    Copies the target Laurent system in double double precision to the systems container for Laurent systems.

phcpy2c3.\texttt{py2c\_copy\_dobldobl\_container\_to\_start\_solutions}()
    Copies the solutions in double double precision from the container to the start solutions in double double precision.

phcpy2c3.\texttt{py2c\_copy\_dobldobl\_container\_to\_start\_system}()
    Copies the system in the container for systems with coefficients in double double precision to the start system.

phcpy2c3.\texttt{py2c\_copy\_dobldobl\_container\_to\_target\_solutions}()
    Copies the solutions in double double precision from the container to the target solutions in double double precision.

phcpy2c3.\texttt{py2c\_copy\_dobldobl\_container\_to\_target\_system}()
    Copies the system in the container for systems with coefficients in double double precision to the target system.
There is one integer parameter dim on input, which represents the dimension of the witness set. Copies the witness set representation for a solution set of dimension dim into the Laurent systems and solutions container, in double double precision. REQUIRED: 1) py2c_dobldobl_laursys_solve was executed successfully, and 2) dim is in the range 0..topdim.

There is one integer parameter dim on input, which represents the dimension of the witness set. Copies the witness set representation for a solution set of dimension dim into the systems and solutions container, in double double precision. REQUIRED: 1) py2c_dobldobl_polysys_solve was executed successfully, and 2) dim is in the range 0..topdim.

Copies the start solutions in double double precision to the container for solutions in double double precision.

Copies the start system to the container for systems with coefficients in double double precision.

Copies the target solutions in double double precision to the container for solutions in double double precision.

Copies the target system to the container for systems with coefficients in double double precision.

Copies the solutions in arbitrary multiprecision from the container to the start solutions in arbitrary multiprecision.

Copies the system in the container for systems with coefficients in arbitrary multiprecision to the start system.

Copies the solutions in arbitrary multiprecision from the container to the target solutions in arbitrary multiprecision.

Copies the system in the container for systems with coefficients in arbitrary multiprecision to the target system.

Copies the start solutions in arbitrary multiprecision to the container for solutions in arbitrary multiprecision.

Copies the start system to the container for systems with coefficients in arbitrary multiprecision.

Copies the target solutions in arbitrary multiprecision to the container for solutions in arbitrary multiprecision.

Copies the target system to the container for systems with coefficients in arbitrary multiprecision.

Copies multiprecision target system to container

Copies the Laurent system in quad double precision from the container to the start system.

Copies the Laurent system in quad double precision from the container to the target system.

Copies the start Laurent system in quad double precision to the systems container for Laurent systems.

Copies the target Laurent system in quad double precision to the systems container for Laurent systems.
phcpy2c3.py2c_copy_quaddobl_container_to_start_solutions()
Copies the solutions in quad double precision from the container to the start solutions in quad double precision.

phcpy2c3.py2c_copy_quaddobl_container_to_start_system()
Copies the system in the container for systems with coefficients in quad double precision to the start system.

phcpy2c3.py2c_copy_quaddobl_container_to_target_solutions()
Copies the solutions in quad double precision from the container to the target solutions in quad double precision.

phcpy2c3.py2c_copy_quaddobl_container_to_target_system()
Copies the system in the container for systems with coefficients in quad double precision to the target system.

phcpy2c3.py2c_copy_quaddobl_laursys_witset()
There is one integer parameter dim on input, which represents the dimension of the witness set. Copies the witness set representation for a solution set of dimension dim into the Laurent systems and solutions container, in quad double precision. REQUIRED: 1) py2c_quaddobl_laursys_solve was executed successfully, and 2) dim is in the range 0..topdim.

phcpy2c3.py2c_copy_quaddobl_polysys_witset()
There is one integer parameter dim on input, which represents the dimension of the witness set. Copies the witness set representation for a solution set of dimension dim into the systems and solutions container, in quad double precision. REQUIRED: 1) py2c_quaddobl_polysys_solve was executed successfully, and 2) dim is in the range 0..topdim.

phcpy2c3.py2c_copy_quaddobl_start_solutions_to_container()
Copies the start solutions in quad double precision to the container for solutions in quad double precision.

phcpy2c3.py2c_copy_quaddobl_start_system_to_container()
Copies the start system to the container for systems with coefficients in quad double precision.

phcpy2c3.py2c_copy_quaddobl_target_solutions_to_container()
Copies the target solutions in quad double precision to the container for solutions in quad double precision.

phcpy2c3.py2c_copy_quaddobl_target_system_to_container()
Copies the target system to the container for systems with coefficients in quad double precision.

phcpy2c3.py2c_copy_standard_Laurent_container_to_start_system()
Copies the Laurent system in standard double precision from the container to the start system.

phcpy2c3.py2c_copy_standard_Laurent_container_to_target_system()
Copies the Laurent system in standard double precision from the container to the target system.

phcpy2c3.py2c_copy_standard_Laurent_start_system_to_container()
Copies the start Laurent system in standard double precision to the systems container for Laurent systems.

phcpy2c3.py2c_copy_standard_Laurent_target_system_to_container()
Copies the target Laurent system in standard double precision to the systems container for Laurent systems.

phcpy2c3.py2c_copy_standard_container_to_start_solutions()
Copies the solutions in standard double precision from the container to the start solutions in standard double precision.

phcpy2c3.py2c_copy_standard_container_to_start_system()
Copies the system in the container for systems with coefficients in standard double precision to the start system.

phcpy2c3.py2c_copy_standard_container_to_target_solutions()
Copies the solutions in standard double precision from the container to the target solutions in standard double precision.

phcpy2c3.py2c_copy_standard_container_to_target_system()
Copies the system in the container for systems with coefficients in standard double precision to the target system.
phcpy2c3.py2c_copy_standard_laursys_witset()
There is one integer parameter dim on input, which represents the dimension of the witness set. Copies the wit-
ness set representation for a solution set of dimension dim into the Laurent systems and solutions container, in standard double precision. REQUIRED: 1) py2c_standard_laursys_solve was executed successfully, and 2) dim is in the range 0..topdim.

phcpy2c3.py2c_copy_standard_polysys_witset()
There is one integer parameter dim on input, which represents the dimension of the witness set. Copies the wit-
ness set representation for a solution set of dimension dim into the systems and solutions container, in standard double precision. REQUIRED: 1) py2c_standard_polysys_solve was executed successfully, and 2) dim is in the range 0..topdim.

phcpy2c3.py2c_copy_standard_target_solutions_to_container()
Copies the target solutions in standard double precision to the container for solutions in standard double precision.

phcpy2c3.py2c_copy_standard_target_system_to_container()
Copies the target system to the container for systems with coefficients in standard double precision.

phcpy2c3.py2c_copy_start_solutions_to_container()
Copies the start solutions in standard double precision to the container for solutions in standard double precision.

phcpy2c3.py2c_copy_start_system_to_container()
Copies the start system to the container for systems with coefficients in standard double precision.

phcpy2c3.py2c_corecount()
Returns the number of cores available for multithreading.

phcpy2c3.py2c_create_dobldobl_homotopy()
Initializes the data for a homotopy in double double precision. The failure code is returned, which is zero when all goes well.

phcpy2c3.py2c_create_dobldobl_homotopy_with_gamma()
Initializes the data for a homotopy in double double precision. On input are two doubles and one positive integer: (1) the real and imaginary part of the gamma constant; (2) the power of t in the homotopy. The failure code is returned, which is zero when all goes well.

phcpy2c3.py2c_create_multprec_homotopy()
Initializes the data for a homotopy in arbitrary multiprecision. The failure code is returned, which is zero when all goes well.

phcpy2c3.py2c_create_multprec_homotopy_with_gamma()
Initializes the data for a homotopy in arbitrary multiprecision. On input are two doubles and one positive integer: (1) the real and imaginary part of the gamma constant; (2) the power of t in the homotopy. The failure code is returned, which is zero when all goes well.

phcpy2c3.py2c_create_quaddobl_homotopy()
Initializes the data for a homotopy in quad double precision. The failure code is returned, which is zero when all goes well.

phcpy2c3.py2c_create_quaddobl_homotopy_with_gamma()
Initializes the data for a homotopy in quad double precision. On input are two doubles and one positive integer: (1) the real and imaginary part of the gamma constant; (2) the power of t in the homotopy. The failure code is returned, which is zero when all goes well.

phcpy2c3.py2c_create_standard_homotopy()
Initializes the data for a homotopy in standard double precision. The failure code is returned, which is zero when all goes well.

phcpy2c3.py2c_create_standard_homotopy_with_gamma()
Initializes the data for a homotopy in standard double precision. On input are two doubles and one positive
integer: (1) the real and imaginary part of the gamma constant; (2) the power of \( t \) in the homotopy. The failure code is returned, which is zero when all goes well.

**phcpy2c3**.py2c_define_output_file()

Prompts the user to define the output file. On return is the failure code, which is zero if all went well.

**phcpy2c3**.py2c_determine_output_during_continuation()

Interactive procedure to determine the level of output during the path tracking.

**phcpy2c3**.py2c_diagonal_symbols_doubler()

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 three integers, \( n, d, nc \), and one string \( s \). On input are \( n \), the ambient dimension = \#variables before the embedding, \( d \) is the number of slack variables, or the dimension of the first set, and in \( s \) (\( nc \) characters) 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.

**phcpy2c3**.py2c_dobldobl_laurent_cascade_homotopy()

Creates a homotopy in double double precision using the stored Laurent systems to go one level down the cascade, removing one slice. On return is the failure code, which equals zero if all went well.

**phcpy2c3**.py2c_dobldobl_Newton_laurent_step()

Applies one Newton step in double double precision to the Laurent system in the standard Laurent systems container and to the solutions in the container. On return is the failure code, which equals zero if all went well.

**phcpy2c3**.py2c_dobldobl_Newton_power_series()

Given in the systems container a polynomial system with coefficients in standard double precision, and in the dobldobl systems pool the leading terms of the power series, this function runs Newton’s method to compute power series solutions of the system in the container, in double double precision. There are four integers on input: 1) the index of the series parameter; 2) the maximal degree of the series; 3) the number of Newton steps to be done on each solution; 4) a 0/1-flag to indicate whether additional diagnostic output needs to be written to screen. On return is the failure code, which equals zero if all went well.

**phcpy2c3**.py2c_dobldobl_Newton_series()

Given in the systems container a polynomial system with coefficients in standard double precision, and in the solutions container the leading coefficients of the power series, this function runs Newton’s method to compute power series solutions of the system in the container, in double double precision. There are four integers on input: 1) the index of the series parameter; 2) the maximal degree of the series; 3) the number of Newton steps to be done on each solution; 4) a 0/1-flag to indicate whether additional diagnostic output needs to be written to screen. On return is the failure code, which equals zero if all went well.

**phcpy2c3**.py2c_dobldobl_Newton_step()

Applies one Newton step in double double precision to the system in the standard systems container and to the solutions in the container. On return is the failure code, which equals zero if all went well.

**phcpy2c3**.py2c_dobldobl_Pade_approximant()

Given in the systems container a polynomial system with coefficients in double double precision, and in the solutions container the leading coefficients of the power series, this function runs Newton’s method to compute power series solutions of the system in the container, in double double precision, followed by the construction of the Pade approximants, for each solution. There are five integers on input: 1) the index of the series parameter; 2) the maximal degree of the series; 3) the number of Newton steps to be done on each solution; 4) a 0/1-flag to indicate whether additional diagnostic output needs to be written to screen. The Pade approximants are stored in the dobldobl systems pool, numerators in the odd indexed entries and denominators in the entries with even index in each system. On return is the failure code, which equals zero if all went well.

**phcpy2c3**.py2c_dobldobl_cascade_homotopy()

Creates a homotopy in double double precision using the stored systems to go one level down the cascade,
removing one slice. On return is the failure code, which equals zero if all went well.

**py2c_dobldobl Collapse Diagonal**

Eliminates the extrinsic diagonal for the system and solutions in the containers for double doubles. On input are two integers: 1) k, the current number of slack variables in the embedding; 2) d, the number of slack variables to add to the final embedding. The system in the container has its diagonal eliminated and is embedded with k+d slack variables. The solutions corresponding to this system are in the solutions container. On return is the failure code, which equals zero if all went well.

**py2c_dobldobl Crude Tracker**

A crude tracker appends the end point of a path directly to the solutions container, without refinement or post-processing. Tracking happens in double double precision. On entry is the verbose parameter which is 1 or 0. If 1, then the solution vectors are written to screen, otherwise the crude tracker stays mute. On return is the failure code, which is zero when all went well. The requirement is that the target system, start system, and start solutions in double double precision have been initialized in the containers.

**py2c_dobldobl Deflate**

Applies deflation in double double precision to the system and the solutions stored in the containers. There are five input parameters, two integers and three doubles: (1) maxitr : the maximum number of iterations per root, (2) maxdef : the maximum number of deflations per root, (3) tolerr : tolerance on the forward error on each root, (4) tolex : tolerance on the backward error on each root, (5) tolres : tolerance on the numerical rank of the Jacobian matrices. On return is the failure code, which equals zero if all went well.

**py2c_dobldobl Diagonal Cascade Solutions**

Makes the start solutions to start the cascade homotopy to intersect two solution sets of dimensions a and b, where a >= b, in double double precision. The dimensions a and b are given as input parameters. The systems stored as target and start system in the container define the witness sets for these two solution sets. On return is the failure code, which equals zero when all went well.

**py2c_dobldobl Diagonal Homotopy**

Creates a diagonal homotopy to intersect two solution sets of dimensions a and b respectively, where a >= b. The two input parameters are values for a and b. The systems stored as target and start system in the container, in double double precision, define the witness sets for these two solution sets.

**py2c_dobldobl Laursys Solve**

Runs the cascades of homotopies on the Laurent polynomial system in the dobldobl systems container. Runs in double double precision. On input are five integers : 1) nbtasks equals the number of tasks for multitasking, 2) topdim is the top dimension to start the homotopy cascades, 3) filter is a 0 or 1 flag to filter the witness supersets, 4) factor is a 0 or 1 flag to factor the witness sets, 5) verbose is a flag for intermediate output.

**py2c_dobldobl Multiplicity Structure**

Computes the multiplicity structure in double double precision. Required is the presence of a polynomial system in the dobldobl systems container and a solution in the dobldobl solutions container. The input parameters are two integers and one double: order : the maximum differentiation order, verbose : 1 for verbose, 0 for silent, and tol : tolerance on the numerical rank. On return is a tuple: the multiplicity and the values of the Hilbert function.

**py2c_dobldobl Polysys Solve**

Runs the cascades of homotopies on the polynomial system in the dobldobl systems container. Runs in double double precision. On input are five integers : 1) nbtasks equals the number of tasks for multitasking, 2) topdim is the top dimension to start the homotopy cascades, 3) filter is a 0 or 1 flag to filter the witness supersets, 4) factor is a 0 or 1 flag to factor the witness sets, 5) verbose is a flag for intermediate output.

**py2c_dobldobl Witset Of Laurent Hypersurface**

Given in the string p of nc characters a polynomial in nv variables, terminated by a semicolon, the systems and solutions container in double double precision on return contain a witness set for the hypersurface defined by the Laurent polynomial in p. On entry are two integers and one string, in the following order: 1) nv, the number of variables of the polynomials; 2) nc, the number of characters in the string p; 3) p, string representation of
a Laurent polynomial in several variables, terminates with a semicolon. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_dobldobl_witset_of_hypersurface()**

Given in the string \( p \) of \( nc \) characters a polynomial in \( nv \) variables, terminated by a semicolon, the systems and solutions container in double double precision on return contain a witness set for the hypersurface defined by the ordinary polynomial in \( p \). On entry are two integers and one string, in the following order: 1) \( nv \), the number of variables of the polynomials; 2) \( nc \), the number of characters in the string \( p \); 3) \( p \), string representation of an ordinary polynomial in several variables, terminates with a semicolon. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_embed_dobldobl_Laurent_system()**

Replaces the Laurent system with coefficients in double double precision in the container with its embedding of dimension \( d \). The dimension \( d \) is given as an integer parameter on input. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_embed_dobldobl_system()**

Replaces the system with coefficients in double double precision in the container with its embedding of dimension \( d \). The dimension \( d \) is given as an integer parameter on input. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_embed_quaddobl_Laurent_system()**

Replaces the Laurent system with coefficients in quad double precision in the container with its embedding of dimension \( d \). The dimension \( d \) is given as an integer parameter on input. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_embed_quaddobl_system()**

Replaces the system with coefficients in quad double precision in the container with its embedding of dimension \( d \). The dimension \( d \) is given as an integer parameter on input. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_embed_standard_Laurent_system()**

Replaces the Laurent system with coefficients in standard double precision in the container with its embedding of dimension \( d \). The dimension \( d \) is given as an integer parameter on input. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_embed_standard_system()**

Replaces the system with coefficients in standard double precision in the container with its embedding of dimension \( d \). The dimension \( d \) is given as an integer parameter on input. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_embed_system()**

Replaces the system in the container with its embedding of dimension \( d \). The dimension \( d \) is given as the first integer parameter on input. The second integer parameter indicates the precision, either 0, 1, or 2, respectively for double, double double, or quad double precision. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_extrinsic_top_diagonal_dimension()**

Returns the dimension of the start and target system to start the extrinsic cascade to intersect two witness sets, respectively of dimensions \( a \) and \( b \), with ambient dimensions respectively equal to \( n1 \) and \( n2 \). There are four integers as parameters on input: \( n1, n2, a \) and \( b \).

**phcpy2c3.py2c_factor_define_output_file_with_string()**

Defines the output file for the factorization. On input are an integer and a string: 1) the integer equals the number of characters in the string; and 2) the string contains the name of a file. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_factor_dobldobl_assign_labels()**

Assigns labels, replacing the multiplicity field of each solution in double double precision stored in the container.
On entry are two integers: 1) \(n\), the number of coordinates of the solutions; 2) \(nbsols\), the number of solutions in the container. On return is the failure code, which equals zero if all went well.

\texttt{phcpy2c3.py2c_factor_dobldobl_trace_grid_diagnostics()}

Returns a tuple of two doubles with the diagnostics on the trace grid computed in double double precision. The first double is the largest error of the samples. The second double is the smallest distance between two samples.

\texttt{phcpy2c3.py2c_factor_dobldobl_trace_sum_difference()}

Returns the difference between the actual sum at the samples defined by the labels to the generic points in the factor, and the trace sum, computed in double double precision. On entry are three integer numbers and one string: 1) \(d\), the number of points in the witness set; 2) \(k\), the dimension of the solution set; 3) \(nc\), the number of characters in the string; 4) \(ws\), the string representing the labels of the witness set.

\texttt{phcpy2c3.py2c_factor_dobldobl_track_paths()}

Tracks as many paths as defined by witness set, in double double precision. On input is an integer, which must be 1 if the witness set is defined by a Laurent polynomial system. On return is the failure code, which is zero when all went well.

\texttt{phcpy2c3.py2c_factor_initialize_dobldobl_Laurent_sampler()}

Initializes the sampling machine with a witness set, defined by a Laurent polynomial system in double double precision. The embedded system is taken from the Laurent systems container and the generic points from the solutions container. On entry is the dimension or the number of hyperplanes to slide the positive dimensional solution set.

\texttt{phcpy2c3.py2c_factor_initialize_dobldobl_monodromy()}

Initializes the internal data structures for \(n\) loops, to factor a \(k\)-dimensional solution component of degree \(d\), in double double precision. There are three integers on input, in the following order: 1) \(n\), the number of loops; 2) \(d\), the degree of the solution set; 3) \(k\), the dimensional of the solution set. On return is the failure code, which equals zero when all went well.

\texttt{phcpy2c3.py2c_factor_initialize_dobldobl_sampler()}

Initializes the sampling machine with a witness set, defined by a Laurent polynomial system in double double precision. The embedded system is taken from the polynomial systems container and the generic points from the solutions container. On entry is the dimension or the number of hyperplanes to slide the positive dimensional solution set.

\texttt{phcpy2c3.py2c_factor_initialize_quaddobl_monodromy()}

Initializes the internal data structures for \(n\) loops, to factor a \(k\)-dimensional solution component of degree \(d\), in quad double precision. There are three integers on input, in the following order: 1) \(n\), the number of loops; 2) \(d\), the degree of the solution set; 3) \(k\), the dimensional of the solution set. On return is the failure code, which equals zero when all went well.

\texttt{phcpy2c3.py2c_factor_initialize_quaddobl_sampler()}

Initializes the sampling machine with a witness set, defined by a Laurent polynomial system in quad double precision. The embedded system is taken from the Laurent systems container and the generic points from the solutions container. On entry is the dimension or the number of hyperplanes to slide the positive dimensional solution set.

\texttt{phcpy2c3.py2c_factor_initialize_standard_Laurent_sampler()}

Initializes the sampling machine with a witness set, defined by a Laurent polynomial system in standard double precision. The embedded system is taken from the Laurent systems container and the generic points from the solutions container. On entry is the dimension or the number of hyperplanes to slide the positive dimensional solution set.
phcpy2c3.py2c_factor_initialize_standard_monodromy()
Initializes the internal data structures for n loops, to factor a k-dimensional solution component of degree d, in standard double precision. There are three integers on input, in the following order: 1) n, the number of loops; 2) d, the degree of the solution set; 3) k, the dimensional of the solution set. On return is the failure code, which equals zero when all went well.

phcpy2c3.py2c_factor_initialize_standard_sampler()
Initializes the sampling machine with a witness set, defined by an ordinary polynomial system in standard double precision. The embedded system is taken from the polynomial systems container and the generic points from the solutions container. On entry is the dimension or the number of hyperplanes to slide the positive dimensional solution set.

phcpy2c3.py2c_factor_new_dobldobl_slices()
Generates k random slides in n-space, in double double precision. The k and the n are the two input parameters. On return is the failure code, which is zero when all went well.

phcpy2c3.py2c_factor_new_quaddobl_slices()
Generates k random slides in n-space, in quad double precision. The k and the n are the two input parameters. On return is the failure code, which is zero when all went well.

phcpy2c3.py2c_factor_new_standard_slices()
Generates k random slides in n-space, in standard double precision. The k and the n are the two input parameters. On return is the failure code, which is zero when all went well.

phcpy2c3.py2c_factor_number_of_dobldobl_components()
Returns the number of irreducible factors in the current double double precision decomposition of the witness set.

phcpy2c3.py2c_factor_number_of_quaddobl_components()
Returns the number of irreducible factors in the current quad double precision decomposition of the witness set.

phcpy2c3.py2c_factor_number_of_standard_components()
Returns the number of irreducible factors in the current standard double precision decomposition of the witness set.

phcpy2c3.py2c_factor_permutation_after_dobldobl_loop()
For a set of degree d, computes the permutation using the solutions most recently stored, after a loop in double double precision. The number d is the input parameter of this function. On return is the string representation of the permutation.

phcpy2c3.py2c_factor_permutation_after_quaddobl_loop()
For a set of degree d, computes the permutation using the solutions most recently stored, after a loop in quad double precision. The number d is the input parameter of this function. On return is the string representation of the permutation.

phcpy2c3.py2c_factor_permutation_after_standard_loop()
For a set of degree d, computes the permutation using the solutions most recently stored, after a loop in standard double precision. The number d is the input parameter of this function. On return is the string representation of the permutation.

phcpy2c3.py2c_factor_quaddobl_assign_labels()
Assigns labels, replacing the multiplicity field of each solution in quad double precision stored in the container. On entry are two integers: 1) n, the number of coordinates of the solutions; 2) nbsols, the number of solutions in the container. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_factor_quaddobl_trace_grid_diagnostics()
Returns a tuple of two doubles with the diagnostics on the trace grid computed in quad double precision. The first double is the largest error of the samples. The second double is the smallest distance between two samples.
phcpy2c3.py2c_factor_quaddobl_trace_sum_difference()  
Returns the difference between the actual sum at the samples defined by the labels to the generic points in the 
factor, and the trace sum, computed in quad double precision. On entry are three integer numbers and one 
string: 1) d, the number of points in the witness set; 2) k, the dimension of the solution set; 3) nc, the number of 
characters in the string; 4) ws, the string representing the labels of the witness set.

phcpy2c3.py2c_factor_quaddobl_track_paths()  
Tracks as many paths as defined by witness set, in quad double precision. On input is an integer, which must be 
1 if the witness set is defined by a Laurent polynomial system. On return is the failure code, which is zero when 
all went well.

phcpy2c3.py2c_factor_restore_dobldobl_solutions()  
Restores the first initialized solutions, in double double precision, from sampler to the container.

phcpy2c3.py2c_factor_restore_quaddobl_solutions()  
Restores the first initialized solutions, in quad double precision, from sampler to the container.

phcpy2c3.py2c_factor_restore_standard_solutions()  
Restores the first initialized solutions, in standard double precision, from sampler to the container.

phcpy2c3.py2c_factor_set_dobldobl_to_mute()  
Sets the state of monodromy permutations in double double precision to silent.

phcpy2c3.py2c_factor_set_dobldobl_to_verbose()  
Sets the state of monodromy permutations in double double precision to verbose.

phcpy2c3.py2c_factor_set_dobldobl_trace_slice()  
Assigns the constant coefficient of the first slice, in double double precision. On entry is a flag to indicate if it 
was the first time or not. On return is the failure code, which is zero if all went well.

phcpy2c3.py2c_factor_set_quaddobl_to_mute()  
Sets the state of monodromy permutations in quad double precision to silent.

phcpy2c3.py2c_factor_set_quaddobl_to_verbose()  
Sets the state of monodromy permutations in quad double precision to verbose.

phcpy2c3.py2c_factor_set_quaddobl_trace_slice()  
Assigns the constant coefficient of the first slice, in quad double precision. On entry is a flag to indicate if it was 
the first time or not. On return is the failure code, which is zero if all went well.

phcpy2c3.py2c_factor_set_standard_to_mute()  
Sets the state of monodromy permutations in standard double precision to silent.

phcpy2c3.py2c_factor_set_standard_to_verbose()  
Sets the state of monodromy permutations in standard double precision to verbose.

phcpy2c3.py2c_factor_set_standard_trace_slice()  
Assigns the constant coefficient of the first slice, in standard double precision. On entry is a flag to indicate if it was 
the first time or not. On return is the failure code, which is zero if all went well.

phcpy2c3.py2c_factor_standard_assign_labels()  
Assigns labels, replacing the multiplicity field of each solution in standard double precision stored in the con-
tainer. On entry are two integers: 1) n, the number of coordinates of the solutions; 2) nbsols, the number of 
solutions in the container. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_factor_standard_trace_grid_diagnostics()  
Returns a tuple of two doubles with the diagnostics on the trace grid computed in standard double precision. The 
first double is the largest error of the samples. The second double is the smallest distance between two samples.

phcpy2c3.py2c_factor_standard_trace_sum_difference()  
Returns the difference between the actual sum at the samples defined by the labels to the generic points in the 
factor, and the trace sum, computed in standard double precision. On entry are three integer numbers and one
string: 1) d, the number of points in the witness set; 2) k, the dimension of the solution set; 3) nc, the number of characters in the string; 4) ws, the string representing the labels of the witness set.

**phcpy2c3.py2c_factor_standard_track_paths()**
Tracks as many paths as defined by witness set, in standard double precision. On input is an integer, which must be 1 if the witness set is defined by a Laurent polynomial system. On return is the failure code, which is zero when all went well.

**phcpy2c3.py2c_factor_store_dobldobl_gammas()**
Stores the gamma constants in double double precision for the sampler in the monodromy loops. Generates as many random complex constants as the value on input. On return is the failure code, which is zero if all went well.

**phcpy2c3.py2c_factor_store_dobldobl_solutions()**
Stores the solutions in the container, in double double precision, to the data for monodromy loops.

**phcpy2c3.py2c_factor_store_quaddobl_gammas()**
Stores the gamma constants in quad double precision for the sampler in the monodromy loops. Generates as many random complex constants as the value on input. On return is the failure code, which is zero if all went well.

**phcpy2c3.py2c_factor_store_quaddobl_solutions()**
Stores the solutions in the container, in quad double precision, to the data for monodromy loops.

**phcpy2c3.py2c_factor_store_standard_gammas()**
Stores the gamma constants in standard double precision for the sampler in the monodromy loops. Generates as many random complex constants as the value on input. On return is the failure code, which is zero if all went well.

**phcpy2c3.py2c_factor_store_standard_solutions()**
Stores the solutions in the container, in standard double precision, to the data for monodromy loops.

**phcpy2c3.py2c_factor_swap_dobldobl_slices()**
Swaps the current slices with new slices and takes new solutions as start to turn back, in double double precision. On return is the failure code, which is zero when all went well.

**phcpy2c3.py2c_factor_swap_quaddobl_slices()**
Swaps the current slices with new slices and takes new solutions as start to turn back, in quad double precision. On return is the failure code, which is zero when all went well.

**phcpy2c3.py2c_factor_swap_standard_slices()**
Swaps the current slices with new slices and takes new solutions as start to turn back, in standard double precision. On return is the failure code, which is zero when all went well.

**phcpy2c3.py2c_factor_update_dobldobl_decomposition()**
Updates the decomposition with the given permutation of d elements, computed in double double precision. On entry are two integers and one string: 1) d, the number of elements in the permutation; 2) nc, the number of characters in the string; 3) p, the string representation of the permutation. Returns one if the current decomposition is certified, otherwise returns zero.

**phcpy2c3.py2c_factor_update_quaddobl_decomposition()**
Updates the decomposition with the given permutation of d elements, computed in quad double precision. On entry are two integers and one string: 1) d, the number of elements in the permutation; 2) nc, the number of characters in the string; 3) p, the string representation of the permutation. Returns one if the current decomposition is certified, otherwise returns zero.

**phcpy2c3.py2c_factor_update_standard_decomposition()**
Updates the decomposition with the given permutation of d elements, computed in standard double precision. On entry are two integers and one string: 1) d, the number of elements in the permutation; 2) nc, the number of characters in the string; 3) p, the string representation of the permutation. Returns one if the current
decomposition is certified, otherwise returns zero.

phcpy2c3.py2c_factor_witness_points_of_dobl_dobl_component()
Returns a string which represents an irreducible component, computed in double double precision. On entry are two integers: 1) the sum of the degrees of all components; 2) the index of the component.

phcpy2c3.py2c_factor_witness_points_of_quad_dobl_component()
Returns a string which represents an irreducible component, computed in quad double precision. On entry are two integers: 1) the sum of the degrees of all components; 2) the index of the component.

phcpy2c3.py2c_factor_witness_points_of_standard_component()
Returns a string which represents an irreducible component, computed in standard double precision. On entry are two integers: 1) the sum of the degrees of all components; 2) the index of the component.

phcpy2c3.py2c_get_default_path_parameters()
Given the working precision (16, 32, or 64), returns the default values of the path parameters, for the path trackers with algorithmic differentiation.

phcpy2c3.py2c_get_gamma_constant()
Returns the gamma constant used by the solve functions. Two integer parameters are expected on input: (1) for the precision, 1, 2, or 4, respectively for double, double double, or quad double precision; and (2) the verbose level. The function returns a tuple of two doubles, for the real and imaginary part of the gamma constant.

phcpy2c3.py2c_get_seed()
Returns the current value of the seed. Using this value in py2c_set_seed will ensure that the results of previous runs can be reproduced.

phcpy2c3.py2c_get_value_of_continuation_parameter()
Returns the value of a continuation parameter. On input is the index of this continuation parameter, an integer ranging from 1 to 34. On return is a double with the value of the corresponding parameter.

phcpy2c3.py2c_giftwrap_clear_3d_facets()
Deallocates list of facets of convex hull stored in 3-space.

phcpy2c3.py2c_giftwrap_clear_4d_facets()
Deallocates list of facets of convex hull stored in 4-space.

phcpy2c3.py2c_giftwrap_clear_support_string()
Deallocates the string representation of the support set that was stored internally by the call py2c_giftwrap_support_size.

phcpy2c3.py2c_giftwrap_convex_hull()
Applies the giftwrapping algorithm to a point configuration. On input are an integer and a string: 1) the number of points in the list; 2) the string representation of a Python list of tuples. When the function returns, the internal data structures to store the convex hull are defined. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_giftwrap_initial_form()
Replaces the system in the Laurent systems container by its initial form. There are three input parameters: 1) the dimension, number of coordinates in the inner normal; 2) the number of characters in the string representation for the normal; 3) the string representation of the inner normal. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_giftwrap_number_of_facets()
Returns the number of facets of the given dimension. On input is an integer, the dimension of the facet.

phcpy2c3.py2c_giftwrap_planar()
Applies the giftwrapping algorithm to a planar point configuration. On input are an integer and a string: 1) the number of points in the list; 2) the string representation of a Python list of tuples. On return is the string representation of the vertex points, sorted so that each two consecutive points define an edge.
phcpy2c3.py2c_giftwrap_retrieve_facet()
Returns the string representation of a facet. On input are two integer numbers: 1) the dimension of the facet; 2) the index of the facet.

phcpy2c3.py2c_giftwrap_support_size()
Returns the number of characters in the string representation of the support of the k-th Laurent polynomial in the container, k is given on input.

phcpy2c3.py2c_giftwrap_support_string()
Returns the string representation of the support of a Laurent polynomial.

phcpy2c3.py2c_initialize_dobldobl_homotopy()
Initializes the homotopy to track a path with a generator, using double double precision arithmetic. There is one integer number on input to be considered as a boolean, as an indicator whether a fixed gamma constant will be used. Before calling this routine the target and start system must be copied over from the dobldobl systems container. The two other input parameters are two doubles: the real and imaginary part of the gamma constant. If the integer parameter equals zero and if the two input doubles are not both zero, then the input gamma constant will be used, otherwise, if the two input doubles are zero and the first integer parameter is zero as well, then a random gamma constant will be generated.

phcpy2c3.py2c_initialize_dobldobl_solution()
Initializes the path tracker with a generator with a solution from the dobldobl solutions container. The index to the solution is given as an integer input parameter. The counting of the indices starts at one, so the first solution has index one.

phcpy2c3.py2c_initialize_multprec_homotopy()
Initializes the homotopy to track a path with a generator, using arbitrary multiprecision arithmetic. There is are two integer numbers on input: 1) one to be considered as a boolean, as an indicator whether a fixed gamma constant will be used; and 2) the number of decimal places in the working precision. Before calling this routine the target and start system must be copied over from the multprec systems container.

phcpy2c3.py2c_initialize_multprec_solution()
Initializes the path tracker with a generator with a solution from the multprec solutions container. The index to the solution is given as an integer input parameter. The counting of the indices starts at one, so the first solution has index one.

phcpy2c3.py2c_initialize_quaddobl_homotopy()
Initializes the homotopy to track a path with a generator, using quad double precision arithmetic. There is one integer number on input to be considered as a boolean, as an indicator whether a fixed gamma constant will be used. Before calling this routine the target and start system must be copied over from the quaddobl systems container. The two other input parameters are two doubles: the real and imaginary part of the gamma constant. If the integer parameter equals zero and if the two input doubles are not both zero, then the input gamma constant will be used, otherwise, if the two input doubles are zero and the first integer parameter is zero as well, then a random gamma constant will be generated.

phcpy2c3.py2c_initialize_quaddobl_solution()
Initializes the path tracker with a generator with a solution from the quaddobl solutions container. The index to the solution is given as an integer input parameter. The counting of the indices starts at one, so the first solution has index one.

phcpy2c3.py2c_initialize_standard_homotopy()
Initializes the homotopy to track a path with a generator, using standard double precision arithmetic. There is one integer number on input to be considered as a boolean, as an indicator whether a fixed gamma constant will be used. Before calling this routine the target and start system must be copied over from the standard systems container. The two other input parameters are two doubles: the real and imaginary part of the gamma constant. If the integer parameter equals zero and if the two input doubles are not both zero, then the input gamma constant will be used, otherwise, if the two input doubles are zero and the first integer parameter is zero as well, then a random gamma constant will be generated.
phcpy2c3.py2c_initialize_standard_solution()
Initializes the path tracker with a generator with a solution from the standard solutions container. The index to
the solution is given as an integer input parameter. The counting of the indices starts at one, so the first solution
has index one.

phcpy2c3.py2c_initialize_varbprec_homotopy()
Initializes the variable precision homotopy with the target and start system stored in the strings. On entry are
three integers and two strings, in the following order: 1) fixed_gamma is a flag: if 1, then a fixed value for
the gamma constant is used, if 0, a random value for gamma will be generated; 2) nc_target, the number of
characters in the string target; 3) target, the string representation of the target system; 4) nc_start, the number of
characters in the string start; 5) start, the string representation of the start system. On return is the failure code,
which equals zero if all went well.

phcpy2c3.py2c_initialize_varbprec_solution()
Uses the string representation of a solution to initialize the variable precision path tracker with. There are three
input parameters, two integers and one string: 1) nv, the number of variables in the solution; 2) nc, the number of
characters in the string sol; 3) sol, the string representation of a solution. On return is the failure code, which
equals zero if all went well.

phcpy2c3.py2c_intcelcon_append_lifted_point()
Appends a lifted point to the cells container. There are three input parameters: 1) the dimension of the point; 2)
the index of the support to where to append to; and 3) the string representation of the lifted point. Returns the
failure code, which equals zero when all went well.

phcpy2c3.py2c_intcelcon_clear_mixed_cell_configuration()
Deallocates the data in the integer cell container.

phcpy2c3.py2c_intcelcon_get_inner_normal()
Given on input the dimension of the lifted points and the index of the mixed cell of interest, returns the string
representation of the inner normal of the mixed cell.

phcpy2c3.py2c_intcelcon_get_lifted_point()
Returns the string representation of the coordinates of a lifted point.

phcpy2c3.py2c_intcelcon_get_point_in_cell()
Returns the string representation of the n coordinates of the k-th point from the j-th list of the i-th cell. On input
are the four integers: n, i, j, k, respectively the length of the lifted vectors in the supports, the index to a cell in
the container, the index to a support of the i-th cell, and the index to a point in the j-th support of the i-th cell.

phcpy2c3.py2c_intcelcon_initialize_supports()
Initializes the supports with an integer valued lifting.

phcpy2c3.py2c_intcelcon_length_of_supports()
Returns a list of lengths of each support.

phcpy2c3.py2c_intcelcon_make_subdivision()
Computes the cells in the regular subdivision induced by an integer valued lifting function.

phcpy2c3.py2c_intcelcon_mixed_volume()
Returns the mixed volume of a mixed cell.

phcpy2c3.py2c_intcelcon_number_of_cells()
Returns the number of cells in the mixed subdivision by integer lifting.

phcpy2c3.py2c_intcelcon_number_of_points_in_cell()
Given are two integer numbers: the index to a cell (starting the count at one) and the number of different
supports. On return is the string representation of the number of points which span each component of the
mixed cell.

phcpy2c3.py2c_intcelcon_read_mixed_cell_configuration()
Reads a mixed-cell configuration
phcpy2c3.py2c_intcelcon_set_type_of_mixture()
Defines the type of mixture of the tuple of supports.

phcpy2c3.py2c_intcelcon_type_of_mixture()
Returns the type of mixture for the integer cells container.

phcpy2c3.py2c_intcelcon_write_mixed_cell_configuration()
Writes the mixed-cell configuration to screen.

phcpy2c3.py2c_linear_reduce_dobldobl_system()
Applies linear reduction on the coefficient matrix of the system in the container for double double precision. There is one integer parameter: whether to diagonalize or not.

phcpy2c3.py2c_linear_reduce_quaddobl_system()
Applies linear reduction on the coefficient matrix of the system in the container for quad double precision. There is one integer parameter: whether to diagonalize or not.

phcpy2c3.py2c_linear_reduce_standard_system()
Applies linear reduction on the coefficient matrix of the system in the container for standard double precision. There is one integer parameter: whether to diagonalize or not.

phcpy2c3.py2c_mapcon_clear_maps()
Deallocates the maps stored in the container. Returns the failure code, which equals zero if all went well.

phcpy2c3.py2c_mapcon_coefficients_of_map()
Returns the coefficients of a monomial map stored in the container. On entry are three parameters: 1) the dimension of the map; 2) the index of the map in all maps of that dimension; 3) the number of variables. On return is a Python list of complex doubles.

phcpy2c3.py2c_mapcon_degree_of_map()
Given the dimension and index of a map, given as two integers as input parameters, returns the degree of that map.

phcpy2c3.py2c_mapcon_exponents_of_map()
Returns the exponents of a monomial map stored in the container. On entry are three parameters: 1) the dimension of the map; 2) the index of the map in all maps of that dimension; 3) the number of variables. On return is a Python list of integers.

phcpy2c3.py2c_mapcon_number_of_maps()
Returns the number of maps in the container.

phcpy2c3.py2c_mapcon_solve_system()
Solves the binomial system stored in the Laurent systems container. There is one input argument, either one or zero. If one, then only the pure top dimensional solutions are computed. If zero, then all solution sets are computed. Returns the failure code, which equals zero if all went well.

phcpy2c3.py2c_mapcon_top_dimension()
Returns the top dimension of the maps in the container.

phcpy2c3.py2c_mapcon_write_maps()
Writes the maps stored in the container to screen. Returns the failure code, which equals zero if all went well.

phcpy2c3.py2c_mixed_volume()
Computes the mixed volume, and the stable mixed volume as well if the input parameter equals 1. On return is the mixed volume, or a tuple with the mixed volume and the stable mixed volume.

phcpy2c3.py2c_mixed_volume_by_demics()
Calls DEMiCs to compute the mixed volume of the system in the standard systems container. If the standard systems container is empty, then the system in the standard Laurent systems container is taken as input. The integer in mv on return equals the mixed volume. The regular mixed-cell configuration is in the cells container.
The above is for the case if the input parameter equals 0. If the input parameter equals 1, then on return is a tuple, which contains the mixed volume and the stable mixed volume.

`phcpy2c3.py2c_multprec_Newton_Laurent_step()`  
Applies one Newton step in arbitrary multiprecision to the Laurent system in the multprec Laurent systems container and to the solutions in the container. On input is an integer: the number of decimal places in the precision. On return is the failure code, which equals zero if all went well.

`phcpy2c3.py2c_multprec_Newton_step()`  
Applies one Newton step in arbitrary multiprecision to the system in the multprec systems container and to the solutions in the container. On input is an integer, the number of decimal places in the precision. On return is the failure code, which equals zero if all went well.

`phcpy2c3.py2c_next_dobldobl_solution()`  
Computes the next point on the solution path with double double precision for the given index. This index is given as an input parameter. The index to the solution path starts its count at one. The point itself is stored in the dobldobl solutions container. The functions `py2c_initialized_dobldobl_tracker` and `py2c_initialize_dobldobl_solution` must have been executed earlier. The failcode is returned, which equals zero if all is well.

`phcpy2c3.py2c_next_multprec_solution()`  
Computes the next point on the solution path with arbitrary multiprecision for the given index. This index is given as an input parameter. The index to the solution path starts its count at one. The point itself is stored in the multprec solutions container. The functions `py2c_initialized_multprec_tracker` and `py2c_initialize_multprec_solution` must have been executed earlier. The failcode is returned, which equals zero if all is well.

`phcpy2c3.py2c_next_quaddobl_solution()`  
Computes the next point on the solution path with quad double precision for the given index. This index is given as an input parameter. The index to the solution path starts its count at one. The point itself is stored in the quaddobl solutions container. The functions `py2c_initialized_quaddobl_tracker` and `py2c_initialize_quaddobl_solution` must have been executed earlier. The failcode is returned, which equals zero if all is well.

`phcpy2c3.py2c_next_standard_solution()`  
Computes the next point on the solution path with standard double precision for the given index. This index is given as an input parameter. The index to the solution path starts its count at one. The point itself is stored in the standard solutions container. The functions `py2c_initialized_standard_tracker` and `py2c_initialize_standard_solution` must have been executed earlier. The failcode is returned, which equals zero if all is well.

`phcpy2c3.py2c_next_varbprec_solution()`  
Computes the next point on a solution path in variable precision. There are four integer input parameters: 1) the number of correct decimal places in the solution; 2) an upper bound on the number of decimal places in the precision; 3) the maximum number of Newton iterations; 4) a flag zero or one to indicate the verbose level. On return is a tuple: 0) the failure code, which equals zero if all went well; and 1) the string representation of the next solution on the path.

`phcpy2c3.py2c_nonlinear_reduce_standard_system()`  
Applies nonlinear reduction on the system in the container for standard double precision. Three integer numbers are expected on input: (1) the maximum number of equal degree replacements, (2) the maximum number of computed S-polynomials, (3) the maximum number of computed R-polynomials. The system in the standard container is replace by the reduced system. Three numbers are returned: (1) the number of equal degree replacements, (2) the number of computed S-polynomials, (3) the number of computed R-polynomials.

`phcpy2c3.py2c_numbtrop_dobldobl_clear()`  
Deallocation of the stored numerically computed tropisms, computed in double double precision.
phcpy2c3.py2c_numbtrop_dobldobl_dimension()

Returns the dimension of the tropisms, stored in double double precision, in the numerical tropisms container.

phcpy2c3.py2c_numbtrop_dobldobl_initialize()

Initializes the numerical tropisms container, in double double precision. The input parameters are nbt : number of tropisms; dim : length_of_each tropism; wnd : winding numbers, as many as nbt; dir : 2*nbt*dim doubles with the coordinates of the tropisms; err : errors on the tropisms, as many doubles as the value of 2*nbt. The numbers in wnd, dir, and err must be given in one string, as the string representation of a list of doubles. On return is the the failure code, which equals zero if all went well.

phcpy2c3.py2c_numbtrop_dobldobl_retrieve()

Retrieves all tropisms stored in double double precision. The input parameters are two integers: nbt : number of tropisms; dim : length_of_each tropism. On return are wnd : winding numbers, as many as nbt; dir : 2*nbt*dim doubles with the coordinates of the tropisms; err : errors on the tropisms, as many doubles as the value of 2*nbt. All numbers are returns in one string, as the string representation of a list of doubles. The failure code, which equals zero if all went well.

phcpy2c3.py2c_numbtrop_dobldobl_retrieve_tropism()

Returns one tropism, stored in double double precision. The input parameters are two integers: dim : the length of the tropism vector; idx : the index of the tropism, indexing starts at one, and ends at nbt, what is returned by numbtrop_dobldobl_size. The first parameter on return is an integer: wnd : estimated winding number; The other output parameters are of type double: dir : coordinates of the tropisms, as many as 2*dim; err : the error on the tropism, two doubles. All 2*dim+2 doubles are returned in one string, the string representation of a list of doubles.

phcpy2c3.py2c_numbtrop_dobldobl_size()

Returns the number of tropisms, stored in double double precision, in the numerical tropisms container.

phcpy2c3.py2c_numbtrop_quaddobl_clear()

Deallocates the stored numerically computed tropisms, computed in quad double precision.

phcpy2c3.py2c_numbtrop_quaddobl_dimension()

Returns the dimension of the tropisms, stored in quad double precision, in the numerical tropisms container.

phcpy2c3.py2c_numbtrop_quaddobl_initialize()

Initializes the numerical tropisms container, in quad double precision. The input parameters are nbt : number of tropisms; dim : length_of_each tropism; wnd : winding numbers, as many as nbt; dir : 4*nbt*dim doubles with the coordinates of the tropisms; err : errors on the tropisms, as many doubles as the value of 4*nbt. The numbers in wnd, dir, and err must be given in one string, as the string representation of a list of doubles. On return is the the failure code, which equals zero if all went well.

phcpy2c3.py2c_numbtrop_quaddobl_retrieve()

Retrieves all tropisms stored in quad double precision. The input parameters are two integers: nbt : number of tropisms; dim : length_of_each tropism. On return are wnd : winding numbers, as many as nbt; dir : 4*nbt*dim doubles with the coordinates of the tropisms; err : errors on the tropisms, as many doubles as the value of 4*nbt. All numbers are returns in one string, as the string representation of a list of doubles. The failure code, which equals zero if all went well.

phcpy2c3.py2c_numbtrop_quaddobl_retrieve_tropism()

Returns one tropism, stored in quad double precision. The input parameters are two integers: dim : the length of the tropism vector; idx : the index of the tropism, indexing starts at one, and ends at nbt, what is returned by numbtrop_quaddobl_size. The first parameter on return is an integer: wnd : estimated winding number; The other output parameters are of type double: dir : coordinates of the tropisms, as many as 4*dim; err : the error on the tropism, four doubles. All 4*dim+4 doubles are returned in one string, the string representation of a list of doubles.

phcpy2c3.py2c_numbtrop_quaddobl_size()

Returns the number of tropisms, stored in quad double precision, in the numerical tropisms container.
Deallocates the stored numerically computed tropisms, computed in standard double precision.

Returns the dimension of the tropisms, stored in standard double precision, in the numerical tropisms container.

Initializes the numerical tropisms container, in standard double precision. The input parameters are nbt : number of tropisms; dim : length_of_each tropism; wnd : winding numbers, as many as nbt; dir : nbt*dim doubles with the coordinates of the tropisms; err : errors on the tropisms, as many doubles as the value of nbt. The numbers in wnd, dir, and err must be given in one string, as the string representation of a list of doubles. On return is the failure code, which equals zero if all went well.

Returns all tropisms stored in standard double precision. The input parameters are two integers: nbt : number of tropisms; dim : length_of_each tropism. On return are wnd : winding numbers, as many as nbt; dir : nbt*dim doubles with the coordinates of the tropisms; err : errors on the tropisms, as many doubles as the value of nbt. All numbers are returns in one string, as the string representation of a list of doubles. The failure code, which equals zero if all went well.

Returns one tropism, stored in standard double precision. The input parameters are two integers: dim : the length of the tropism vector; idx : the index of the tropism, indexing starts at one, and ends at nbt, what is returned by numbtrop_standard_size. The first parameter on return is an integer: wnd : estimated winding number; The other output parameters are of type double: dir : coordinates of the tropisms, as many as dim; err : the error on the tropism. All dim+1 doubles are returned in one string, the string representation of a list of doubles.

Returns the number of tropisms, stored in standard double precision, in the numerical tropisms container.

Stores a tropism given in double double precision. The first three input parameters are integers: dim : the length of the tropism vector; idx : the index of the tropism, indexing starts at one, and ends at nbt, what is returned by dobldobl_size; wnd : estimated winding number; The other input parameters are of type double: dir : coordinates of the tropisms, as many as 2*dim; err : the error on the tropism, two doubles. All 2*dim+2 doubles are given in one string, the string representatin of a list of doubles.

Stores a tropism given in quad double precision. The first three input parameters are integers: dim : the length of the tropism vector; idx : the index of the tropism, indexing starts at one, and ends at nbt, what is returned by quaddobl_size; wnd : estimated winding number; dir : coordinates of the tropisms, as many as 4*dim; err : the error on the tropism, four double All 4*dim+4 doubles are given in one string, the string representatin of a list of doubles.

Stores a tropism given in standard double precision. The first three input parameters are integers: dim : the length of the tropism vector; idx : the index of the tropism, indexing starts at one, and ends at nbt, what is returned by standard_size; wnd : estimated winding number; The other input parameters are of type double: dir : coordinates of the tropisms, as many as dim; err : the error on the tropism. All dim+1 doubles are given in one string, the string representation of a list of doubles.

Deallocates data for the series-Pade tracker in double double precision.

Deallocates the allocated space for the parameters.

Deallocates data for the series-Pade tracker in quad double precision.
Deallocates data for the series-Pade tracker in double precision.

Returns the complex number representation of the closest pole, computed by the predictor in double double precision. The returned numbers are the high parts of the double doubles. Results are meaningful only if the real part >= 0.0.

Returns a tuple: the real and imaginary parts of the coefficient of the denominator of the Pade approximant, at the component with leadidx at position idx, computed by the predictor in double double precision. The doubles are the highest parts of the double doubles. The integers leadidx and idx are two input parameters, the third input integer is the verbose flag.

Returns the estimated distance to the closest solution by the path tracker which runs in double double precision.

Returns the current Hessian step size of the path tracker which runs in double double precision.

For the defined target and start system, initializes the homotopy in double double precision, for the step-by-step Pade continuation. On entry is one parameter, the verbose flag which is zero or one. If the verbose flag is 1, then extra output will be written.

On entry are two integers: 1) the index for the continuation parameter in the natural homotopy and 2) the verbose flag. With the system, defined as target system, and the index for the continuation parameter, initializes the homotopy in double double precision for the step-by-step Pade continuation. If the verbose flag is 1, then extra output will be written.

Returns a tuple: the real and imaginary parts of the pole Pade approximant with leadidx at position poleidx, computed by the predictor in double double precision. The integers leadidx and poleidx are two input parameters, the third input integer is the verbose flag. The returned doubles are the highest parts of the double doubles.

Returns the smallest pole radius computed by the predictor in double double precision. The returned number is the high part of the double double number.

Returns the current pole step size of the path tracker which runs in double double precision.

Executes one predict-correct step on the current solution and the defined homotopy in double double precision. On entry is one integer, the verbose flag which is zero or one. On return is the failure code of the predict-correct step: if zero, then the required accuracies were met, otherwise, either the predict or the correct step failed. If the verbose flag is 1, then extra output will be written.

Returns a tuple: the real and imaginary parts of the series coefficient of component with leadidx at position idx, of the series computed by the predictor in double double precision. The doubles are the highest parts of the double doubles. The integers leadidx and idx are two input parameters, the third input integer is the verbose flag.
phcpy2c3.\texttt{py2c\_padcon\_dobldobl\_series\_step}()

Returns the current series step size of the path tracker which runs in double double precision.

phcpy2c3.\texttt{py2c\_padcon\_dobldobl\_step\_size}()

Returns the current step size of the path tracker which runs in double double precision.

phcpy2c3.\texttt{py2c\_padcon\_dobldobl\_t\_value}()

Returns the current t value of the path tracker which runs in double double precision.

phcpy2c3.\texttt{py2c\_padcon\_dobldobl\_track}()

For the defined target, start system, and start solutions, launches the Pade continuation in double double precision. Seven input parameters are expected: 1) the number of characters in the name of the output file; 2) a string which defines the name of the output file, if the string is empty, then no file is created; 3) a flag to indicate whether the output file is the defined output file (value 1 of the flag), or whether the file is local (value 0); 4) an integer for the verbose flag, if zero, then no extra information is written to file or screen; 5) an integer for the homogenization, if zero, tracking happens in affine space, if one, then tracking happens in 1-projective space, if m, for m > 1, then multihomogenization is applied; 6) an integer for the number of variables, 0 if the fifth parameter m is zero or one; 7) a string with the index representation for the partition of the set of variables, if the fifth parameter m is larger than one.

phcpy2c3.\texttt{py2c\_padcon\_get\_dobldobl\_solution}()

On entry are two integer parameters: 1) the index of the position of the solution and 2) the verbose flag, which is zero or one. Retrieves the current solution and places it at the given position in the solutions container in double double precision. If the verbose flag is 1, then extra output will be written.

phcpy2c3.\texttt{py2c\_padcon\_get\_homotopy\_continuation\_parameter}()

Returns the value of the k-th continuation parameter, if k ranges between 1 and 13. The integer k is given on entry.

phcpy2c3.\texttt{py2c\_padcon\_get\_quaddobl\_solution}()

On entry are two integer parameters: 1) the index of the position of the solution and 2) the verbose flag, which is zero or one. Retrieves the current solution and places it at the given position in the solutions container in quad double precision. If the verbose flag is 1, then extra output will be written.

phcpy2c3.\texttt{py2c\_padcon\_get\_standard\_solution}()

On entry are two integer parameters: 1) the index of the position of the solution and 2) the verbose flag, which is zero or one. Retrieves the current solution and places it at the given position in the solutions container in standard double precision. If the verbose flag is 1, then extra output will be written.

phcpy2c3.\texttt{py2c\_padcon\_initialize\_dobldobl\_solution}()

Takes the solution with a given index in the solutions container in double double precision and initializes the series-Pade tracker. On entry are two integers: 1) the index of the position of the solution in the container and 2) the verbose flag, which is zero or one. If the verbose flag is 1, then extra output will be written.

phcpy2c3.\texttt{py2c\_padcon\_initialize\_quaddobl\_solution}()

Takes the solution with a given index in the solutions container in quad double precision and initializes the series-Pade tracker. On entry are two integers: 1) the index of the position of the solution in the container and 2) the verbose flag, which is zero or one. If the verbose flag is 1, then extra output will be written.

phcpy2c3.\texttt{py2c\_padcon\_initialize\_standard\_solution}()

Takes the solution with a given index in the solutions container in standard double precision and initializes the series-Pade tracker. On entry are two integers: 1) the index of the position of the solution in the container and 2) the verbose flag, which is zero or one. If the verbose flag is 1, then extra output will be written.

phcpy2c3.\texttt{py2c\_padcon\_quaddobl\_closest\_pole}()

Returns the complex number representation of the closest pole, computed by the predictor in quad double precision. The returned numbers are the highest parts of the quad doubles. Results are meaningful only if the real part $\geq 0.0$. 

The module `phcpy2c3` contains several functions for computing coefficients, distances, and steps in the Pade approximants and homotopy continuation in quad double precision. Here are brief descriptions of each function:

- `py2c_padcon_quaddobl_denominator_coefficient()`: Returns a tuple: the real and imaginary parts of the series coefficient of the denominator of the Pade approximant, at the component with leadidx at position idx, computed by the predictor in quad double precision. The doubles are the highest parts of the quad doubles. The integers leadidx and idx are two input parameters, the third input integer is the verbose flag.

- `py2c_padcon_quaddobl_estimated_distance()`: Returns the estimated distance to the closest solution by the path tracker which runs in quad double precision.

- `py2c_padcon_quaddobl_hessian_step()`: Returns the current Hessian step size of the path tracker which runs in quad double precision.

- `py2c_padcon_quaddobl_initialize_homotopy()`: For the defined target and start system, initializes the homotopy in quad double precision, for the step-by-step Pade continuation. On entry is one parameter, the verbose flag which is zero or one. If the verbose flag is 1, then extra output will be written.

- `py2c_padcon_quaddobl_initialize_parameter_homotopy()`: On entry are two integers: 1) the index for the continuation parameter in the natural homotopy and 2) the verbose flag. With the system, defined as target system, and the index for the continuation parameter, initializes the homotopy in quad double precision for the step-by-step Pade continuation. If the verbose flag is 1, then extra output will be written.

- `py2c_padcon_quaddobl_numerator_coefficient()`: Returns a tuple: the real and imaginary parts of the series coefficient of the numerator of the Pade approximant, at the component with leadidx at position idx, computed by the predictor in quad double precision. The doubles are the highest parts of the quad doubles. The integers leadidx and idx are two input parameters, the third input integer is the verbose flag.

- `py2c_padcon_quaddobl_pole()`: Returns a tuple: the real and imaginary parts of the pole Pade approximant with leadidx at position poleidx, computed by the predictor in quad double precision. The integers leadidx and poleidx are two input parameters, the third input integer is the verbose flag. The returned doubles are the highest parts of the quad doubles.

- `py2c_padcon_quaddobl_pole_radius()`: Returns the smallest pole radius computed by the predictor in quad double precision. The returned number is the highest part of the quad double number.

- `py2c_padcon_quaddobl_pole_step()`: Returns the current pole step size of the path tracker which runs in quad double precision.

- `py2c_padcon_quaddobl_predict_correct()`: Executes one predict-correct step on the current solution and the defined homotopy in quad double precision. On entry is one integer, the verbose flag which is zero or one. On return is the failure code of the predict-correct step: if zero, then the required accuracies were met, otherwise, either the predict or the correct step failed. If the verbose flag is 1, then extra output will be written.

- `py2c_padcon_quaddobl_series_coefficient()`: Returns a tuple: the real and imaginary parts of the series coefficient of component with leadidx at position idx, of the series computed by the predictor in quad double precision. The doubles are the highest parts of the quad doubles. The integers leadidx and idx are two input parameters, the third input integer is the verbose flag.

- `py2c_padcon_quaddobl_series_step()`: Returns the current series step size of the path tracker which runs in quad double precision.

- `py2c_padcon_quaddobl_t_value()`: Returns the current t value of the path tracker which runs in quad double precision.
For the defined target, start system, and start solutions, launches the Pade continuation in quad double precision. Seven input parameters are expected: 1) the number of characters in the name of the output file; 2) a string which defines the name of the output file, if the string is empty, then no file is created; 3) a flag to indicate whether the output file is the defined output file (value 1 of the flag), or whether the file is local (value 0); 4) an integer for the verbose flag, if zero, then no extra information is written to file or screen; 5) an integer for the homogenization, if zero, tracking happens in affine space, if one, then tracking happens in 1-projective space, if m, for m > 1, then multihomogenization is applied; 6) an integer for the number of variables, 0 if the fifth parameter m is zero or one; 7) a string with the index representation for the partition of the set of variables, if the fifth parameter m is larger than one.

Sets the default values of the homotopy continuation parameters.

The gamma constant is the first homotopy continuation parameter. The gamma is a complex number and it should be given as two doubles, as its real and imaginary part respectively.

Sets the value of the k-th continuation parameter to the given value. The first parameter k is an integer number between 2 and 13. The second parameter is the value of the k-th parameter, parsed as a floating point number.

Returns the complex number representation of the closest pole, computed by the predictor in standard double precision. Results are meaningful only if the real part >= 0.0.

Returns a tuple: the real and imaginary parts of the coefficient of the denominator of the Pade approximant, at the component with leadidx at position idx, computed by the predictor in double precision. The integers leadidx and idx are two input parameters, the third input integer is the verbose flag.

Returns the estimated distance to the closest solution by the path tracker which runs in standard double precision.

Returns the current Hessian step size of the path tracker which runs in standard double precision.

For the defined target and start system, initializes the homotopy in standard double precision, for the step-by-step Pade continuation. On entry is one parameter, the verbose flag which is zero or one. If the verbose flag is 1, then extra output will be written.

On entry are two integers: 1) the index for the continuation parameter in the natural homotopy and 2) the verbose flag. With the system, defined as target system, and the index for the continuation parameter, initializes the homotopy in standard double precision for the step-by-step Pade continuation. If the verbose flag is 1, then extra output will be written.

Returns a tuple: the real and imaginary parts of the coefficient of the numerator of the Pade approximant, at the component with leadidx at position idx, computed by the predictor in double precision. The integers leadidx and idx are two input parameters, the third input integer is the verbose flag.

Returns a tuple: the real and imaginary parts of the pole Pade approximant with leadidx at position poleidx, computed by the predictor in double precision. The integers leadidx and poleidx are two input parameters, the third input integer is the verbose flag.

Returns the smallest pole radius computed by the predictor in standard double precision.
phcpy2c3.py2c_padcon_standard_pole_step()
   Returns the current pole step size of the path tracker which runs in standard double precision.

phcpy2c3.py2c_padcon_standard_predict_correct()
   Executes one predict-correct step on the current solution and the defined homotopy in standard double precision.
   On entry is one integer, the verbose flag which is zero or one. On return is the failure code of the predict-correct
   step: if zero, then the required accuracies were met, otherwise, either the predict or the correct step failed. If the
   verbose flag is 1, then extra output will be written.

phcpy2c3.py2c_padcon_standard_series_coefficient()
   Returns a tuple: the real and imaginary parts of the series coefficient of component with leadidx at position
   idx, of the series computed by the predictor in double precision. The integers leadidx and idx are two input
   parameters, the third input integer is the verbose flag.

phcpy2c3.py2c_padcon_standard_series_step()
   Returns the current series step size of the path tracker which runs in standard double precision.

phcpy2c3.py2c_padcon_standard_step_size()
   Returns the current step size of the path tracker which runs in standard double precision.

phcpy2c3.py2c_padcon_standard_t_value()
   Returns the current t value of the path tracker which runs in standard double precision.

phcpy2c3.py2c_padcon_standard_track()
   For the defined target, start system, and start solutions, launches the Pade continuation in standard double
   precision. Seven input parameters are expected: 1) the number of characters in the name of the output file; 2) a
   string which defines the name of the output file, if the string is empty, then no file is created; 3) a flag to indicate
   whether the output file is the defined output file (value 1 of the flag), or whether the file is local (value 0); 4) an
   integer for the verbose flag, if zero, then no extra information is written to file or screen; 5) an integer for the
   homogenization, if zero, tracking happens in affine space, if one, then tracking happens in 1-projective space,
   if m, for m > 1, then multihomogenization is applied; 6) an integer for the number of variables, 0 if the fifth
   parameter m is zero or one; 7) a string with the index representation for the partition of the set of variables, if
   the fifth parameter m is larger than one.

phcpy2c3.py2c_product_clear_set_structure()
   Deallocates the set structure.

phcpy2c3.py2c_product_is_set_structure_supporting()
   Checks whether the stored set structure is supporting for the system in the standard systems container. Returns
   an integer which represents true (1) or false (0).

phcpy2c3.py2c_product_linear_product_root_count()
   Returns the linear-product root count, computed from the supporting set structure.

phcpy2c3.py2c_product_m_homogeneous_Bezout_number()
   For the system in the standard systems container, a heuristic partition of the set of variables may lead to a
   Bezout number that is smaller than the total degree. On return is the m-homogeneous Bezout number for the
   string representation of the partition that is returned as the second argument in the tuple.

phcpy2c3.py2c_product_m_homogeneous_start_system()
   Given a partition of the set of variables, constructs an m-homogeneous Bezout number for the system in the
   standard systems container. On input are two arguments: 1) the number of characters in the string (second
   argument); and 2) the string representation for a partition of the variables. On return is the m-homogeneous
   Bezout number.

phcpy2c3.py2c_product_m_partition_Bezout_number()
   Given a partition of the set of variables, computes the m-homogeneous Bezout number for the system in the
   standard systems container. On input are two arguments: 1) the number of characters in the string (second
   argument); and 2) the string representation for a partition of the variables. On return is the m-homogeneous
   Bezout number.
phcpy2c3. **py2c_product_parse_set_structure()**

Parses a given string into a set structure. On input are two parameters, one integer and one string: 1) the number of characters in the given string; and 2) the characters in the string. On return is the failure code, if zero, then the string has been parsed into a valid set structure.

phcpy2c3. **py2c_product_random_linear_product_system()**

Builds a random linear-product system based on the stored set structure. On return is the failure code, which equals zero if all went well.

phcpy2c3. **py2c_product_set_structure_string()**

Returns the string representation of the set structure.

phcpy2c3. **py2c_product_solve_linear_product_system()**

Computes all solutions to the linear-product system and stores the solutions in the container for solutions in standard double precision. On return is the failure code, which equals zero if all went well.

phcpy2c3. **py2c_product_supporting_set_structure()**

Builds a supporting set structure for the system stored in the container with coefficients in standard double precision.

phcpy2c3. **py2c_product_write_set_structure()**

Writes the supporting set structure to screen.

phcpy2c3. **py2c_quaddobl_Laurent_cascade_homotopy()**

Creates a homotopy in quad double precision using the stored Laurent systems to go one level down the cascade, removing one slice. On return is the failure code, which equals zero if all went well.

phcpy2c3. **py2c_quaddobl_Newton_Laurent_step()**

Applies one Newton step in quad double precision to the Laurent system in the standard Laurent systems container and to the solutions in the container. On return is the failure code, which equals zero if all went well.

phcpy2c3. **py2c_quaddobl_Newton_power_series()**

Given in the systems container a polynomial system with coefficients in standard double precision, and in the quaddobl systems pool the leading terms of the power series, this function runs Newton’s method to compute power series solutions of the system in the container, in quad double precision. There are four integers on input: 1) the index of the series parameter; 2) the maximal degree of the series; 3) the number of Newton steps to be done on each solution; 4) a 0/1-flag to indicate whether additional diagnostic output needs to be written to screen. On return is the failure code, which equals zero if all went well.

phcpy2c3. **py2c_quaddobl_Newton_series()**

Given in the systems container a polynomial system with coefficients in standard double precision, and in the solutions container the leading coefficients of the power series, this function runs Newton’s method to compute power series solutions of the system in the container, in quad double precision. There are four integers on input: 1) the index of the series parameter; 2) the maximal degree of the series; 3) the number of Newton steps to be done on each solution; 4) a 0/1-flag to indicate whether additional diagnostic output needs to be written to screen. On return is the failure code, which equals zero if all went well.

phcpy2c3. **py2c_quaddobl_Newton_step()**

Applies one Newton step in quad double precision to the system in the standard systems container and to the solutions in the container. On return is the failure code, which equals zero if all went well.

phcpy2c3. **py2c_quaddobl_Pade_approximant()**

Given in the systems container a polynomial system with coefficients in quad double precision, and in the solutions container the leading coefficients of the power series, this function runs Newton’s method to compute power series solutions of the system in the container, in quad double precision, followed by the construction of the Pade approximants, for each solution. There are five integers on input: 1) the index of the series parameter; 2) the degree of the numerator of the Pade approximant; 3) the degree of the denominator of the Pade approximant; 4) the number of Newton steps to be done on each solution; 5) a 0/1-flag to indicate whether additional diagnostic output needs to be written to screen. The Pade approximants are stored in the quaddobl systems pool, numerators
in the odd indexed entries and denominators in the entries with even index in each system. On return is the failure
code, which equals zero if all went well.

phcpy2c3.py2c_quaddobl_cascade_homotopy()
Creates a homotopy in quad double precision using the stored systems to go one level down the cascade, remov-
ing one slice. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_quaddoblCollapse_diagonal()
Eliminates the extrinsic diagonal for the system and solutions in the containers for quad doubles. On input are
two integers: 1) k, the current number of slack variables in the embedding; 2) d, the number of slack variables
to add to the final embedding. The system in the container has its diagonal eliminated and is embedded with
k+d slack variables. The solutions corresponding to this system are in the solutions container. On return is the
failure code, which equals zero if all went well.

phcpy2c3.py2c_quaddobl_crude_tracker()
A crude tracker appends the end point of a path directly to the solutions container, without refinement or post-
processing. Tracking happens in quad double precision. On entry is the verbose parameter which is 1 or 0.
If 1, then the solution vectors are written to screen, otherwise the crude tracker stays mute. On return is the
failure code, which is zero when all went well. The requirement is that the target system, start system, and start
solutions in quad double precision have been initialized in the containers.

phcpy2c3.py2c_quaddobl_deflate()
Applies deflation in quad double precision to the system and the solutions stored in the containers. There are
five input parameters, two integers and three doubles: (1) maxitr : the maximum number of iterations per root,
(2) maxdef : the maximum number of deflations per root, (3) toller : tolerance on the forward error on each
root, (4) tolres : tolerance on the backward error on each root, (5) tolres : tolerance on the numerical rank of the
Jacobian matrices. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_quaddobl_diagonal_cascade_solutions()
Makes the start solutions to start the cascade homotopy to intersect two solution sets of dimensions a and b,
where a >= b, in quad double precision. The dimensions a and b are given as input parameters. The systems
stored as target and start system in the container define the witness sets for these two solution sets. On return is
the failure code, which equals zero when all went well.

phcpy2c3.py2c_quaddobl_diagonal_homotopy()
Creates a diagonal homotopy to intersect two solution sets of dimensions a and b respectively, where a >= b.
The two input parameters are values for a and b. The systems stored as target and start system in the container,
in quad double precision, define the witness sets for these two solution sets.

phcpy2c3.py2c_quaddobl_laursys_solve()
Runs the cascades of homotopies on the Laurent polynomial system in the quaddobl systems container. Runs
in quad double precision. On input are five integers : 1) nbtasks equals the number of tasks for multitasking, 2)
topdim is the top dimension to start the homotopy cascades, 3) filter is a 0 or 1 flag to filter the witness supersets,
4) factor is a 0 or 1 flag to factor the witness sets, 5) verbose is a flag for intermediate output.

phcpy2c3.py2c_quaddobl_multiplicity_structure()
Computes the multiplicity structure in quad double precision. Required is the presence of a polynomial system
in the quaddobl systems container and a solution in the quaddobl solutions container. The input parameters are
two integers and one double: order : the maximum differentiation order, verbose : 1 for verbose, 0 for silent,
and tol : tolerance on the numerical rank. On return is a tuple: the multiplicity and the values of the Hilbert
function.

phcpy2c3.py2c_quaddobl_polysys_solve()
Runs the cascades of homotopies on the polynomial system in the quaddobl systems container. Runs in quad
double precision. On input are five integers : 1) nbtasks equals the number of tasks for multitasking, 2) topdim
is the top dimension to start the homotopy cascades, 3) filter is a 0 or 1 flag to filter the witness supersets, 4)
factor is a 0 or 1 flag to factor the witness sets, 5) verbose is a flag for intermediate output.
Given in the string p of nc characters a polynomial in nv variables, terminated by a semicolon, the systems and solutions container in quad double precision on return contain a witness set for the hypersurface defined by the Laurent polynomial in p. On entry are two integers and one string, in the following order: 1) nv, the number of variables of the polynomials; 2) nc, the number of characters in the string p; 3) p, string representation of a Laurent polynomial in several variables, terminates with a semicolon. On return is the failure code, which equals zero if all went well.

Given in the string p of nc characters a polynomial in nv variables, terminated by a semicolon, the systems and solutions container in quad double precision on return contain a witness set for the hypersurface defined by the ordinary polynomial in p. On entry are two integers and one string, in the following order: 1) nv, the number of variables of the polynomials; 2) nc, the number of characters in the string p; 3) p, string representation of an ordinary polynomial in several variables, terminates with a semicolon. On return is the failure code, which equals zero if all went well.
phcpy2c3.py2c_read_quaddobl_target_system()
Prompts the user to enter a target system that will be parsed in quad double precision. The failure code is returned, which is zero if all went well.

phcpy2c3.py2c_read_quaddobl_target_system_from_file()
The two input arguments are a number and a string: 1) The number equals the number of characters in the string.
2) The string given on input is the name of a file which contains a target system to be parsed in quad double precision. The failure code is returned, which is zero if all went well.

phcpy2c3.py2c_read_standard_start_Laurent_system()
Prompts the user for a file name and reads the start system from file, in standard double precision. If available on file, also its solutions will be read and stored.

phcpy2c3.py2c_read_standard_start_system()
Prompts the user to enter a start system that will be parsed in standard double precision. The failure code is returned, which is zero if all went well.

phcpy2c3.py2c_read_standard_start_system_from_file()
The two input arguments are a number and a string: 1) The number equals the number of characters in the string.
2) The string given on input is the name of a file which contains a start system to be parsed in standard double precision. The failure code is returned, which is zero if all went well.

phcpy2c3.py2c_read_standard_target_Laurent_system()
Prompts the user for a file name and reads the target system from file, in standard double precision. If available on file, also its solutions will be read and stored.

phcpy2c3.py2c_read_standard_target_system()
Prompts the user to enter a target system that will be parsed in standard double precision. The failure code is returned, which is zero if all went well.

phcpy2c3.py2c_read_standard_target_system_from_file()
The two input arguments are a number and a string: 1) The number equals the number of characters in the string.
2) The string given on input is the name of a file which contains a target system to be parsed in standard double precision. The failure code is returned, which is zero if all went well.

phcpy2c3.py2c_scale_dobldobl_solutions()
Replaces the solutions in the dobldobl solutions container with the scaled solutions, scaled with double double precision arithmetic, using the given scaling coefficients. On entry are two parameters: an integer and a string. The integer contains the number of elements in the list of scaling coefficients (doubles) stored in the string. The format of the string is the Python string representation of a list of doubles, i.e.: starting with [ and ending with ].

phcpy2c3.py2c_scale_dobldobl_system()
Applies scaling to the system in the dobldobl systems container, with double double precision arithmetic. The system in the dobldobl systems container is replaced by the scaled system. On entry is one integer, which should be either 0, 1, or 2: 0 for only scaling of the equations, 1 variable scaling without variability reduction, 2 variable scaling with variability reduction. On return is a tuple with the scaling coefficients (if mode > 0) and the estimated inverse condition number of the scaling problem.

phcpy2c3.py2c_scale_quaddobl_solutions()
Replaces the solutions in the quaddobl solutions container with the scaled solutions, scaled with quad double precision arithmetic, using the given scaling coefficients. On entry are two parameters: an integer and a string. The integer contains the number of elements in the list of scaling coefficients (doubles) stored in the string. The format of the string is the Python string representation of a list of doubles, i.e.: starting with [ and ending with ].

phcpy2c3.py2c_scale_quaddobl_system()
Applies scaling to the system in the quaddobl systems container, with quad double precision arithmetic. The system in the quaddobl systems container is replaced by the scaled system. On entry is one integer, which should be either 0, 1, or 2: 0 for only scaling of the equations, 1 variable scaling without variability reduction,
2 variable scaling with variability reduction. On return is a tuple with the scaling coefficients (if mode > 0) and the estimated inverse condition number of the scaling problem.

**phcpy2c3.py2c_scale_standard_solutions()**
Replaces the solutions in the standard solutions container with the scaled solutions, scaled with standard double precision arithmetic, using the given scaling coefficients. On entry are two parameters: an integer and a string. The integer contains the number of elements in the list of scaling coefficients (doubles) stored in the string. The format of the string is the Python string representation of a list of doubles, i.e.: starting with [ and ending with ].

**phcpy2c3.py2c_scale_standard_system()**
Applies scaling to the system in the standard systems container, with standard double precision arithmetic. The system in the standard systems container is replaced by the scaled system. On entry is one integer, which should be either 0, 1, or 2: 0 for only scaling of the equations, 1 variable scaling without variability reduction, 2 variable scaling with variability reduction. On return is a tuple with the scaling coefficients (if mode > 0) and the estimated inverse condition number of the scaling problem.

**phcpy2c3.py2c_scan_for_symbols()**
Given on input are two arguments: a number and a string. The string holds the string representation of a polynomial system, where each polynomial is terminated by a semi colon. The first argument on input is the number of characters in the string. On return is the number of symbols used as variables in the system. This function helps to determine whether a system is square or not.

**phcpy2c3.py2c_schubert_dobldobl_littlewood_richardson_homotopies()**
Runs the Littlewood-Richardson homotopies to resolve a number of general Schubert intersection conditions on k-planes in n-space, in double double precision. The polynomial system that was solved is in the container for systems with coefficients in double double precision and the corresponding solutions are in the dobldobl solutions container. On entry are seven integers and two strings, in the following order: 1) n, the ambient dimension, where the k-planes live; 2) k, the dimension of the solution planes; 3) c, the number of intersection conditions; 4) nc, the number of characters in the string brackets; 5) brackets is a string representation of c brackets, where the numbers in each bracket are separated by spaces; 6) the flag verbose: if 0, then no intermediate output is written, if 1, then the resolution is displayed on screen; 7) the flag verify: if 0, then no diagnostic verification is done, if 1, then diagnostic verification is written to file; 8) the flag minrep: if 0, then all minors are used in the system, if 1, then a minimal representation of the problem is used; 9) the flag tosquare: if 0, then Gauss-Newton path trackers run, if 1, then the overdetermined systems are squared; 10) nbchar, the number of characters in the string filename; 11) filename is the name of the output file. The function returns a tuple of an integer and a string: 0) r is the formal root count as the number of k-planes for conditions imposed by the brackets for general flags; 1) flags, a string with the coefficients of the general flags.

**phcpy2c3.py2c_schubert_localization_poset()**
Returns the string representation of the localization poset for the Pieri root count for m, p, and q. The input parameters are the integer values for m, p, and q: 1) m, the dimension of the input planes; 2) p, the dimension of the output planes; 3) q, the degree of the curves that produce p-planes.

**phcpy2c3.py2c_schubert_osculating_planes()**
Returns the string representation of n real m-planes in d-space osculating a rational normal curve at the n points in s, where n = m*p + q*(m+p) and d = m+p. On entry are four integers and one string: 1) m, the dimension of the input planes; 2) p, the dimension of the output planes; 3) q, the degree of the solution maps; 4) nc, the number of characters in the string pts; and 5) pts, the string with m*p + q*(m+p) interpolation points.

**phcpy2c3.py2c_schubert_pieri_count()**
Returns the number of p-plane producing curves of degree q that meet m*p + q*(m+p) given general m-planes. On input are three integer numbers: 1) m, the dimension of the input planes; 2) p, the dimension of the output planes; and 3) q, the degree of the curve that produces p-planes. The dimension of the ambient space of this Pieri problem is m+p.

**phcpy2c3.py2c_schubert_pieri_homotopies()**
Runs the Pieri homotopies for (m,p,q) dimensions on generic input data. On return the systems container for systems with coefficients in standard double precision contains the polynomial system solved and in the solutions
in standard double precision are in the solutions container. On entry are four integers and two strings: 1) m, the
dimension of the input planes; 2) p, the dimension of the output planes; 3) q, the degree of the solution maps;
4) nc, the number of characters in the string A; 5) A, the string with m*p + q*(m+p) random complex input
m-planes, where the real and imaginary parts are separated by a space; 6) pts, the string with m*p + q*(m+p)
random complex interpolation points, only needed if q > 0. The function returns the combinatorial Pieri root
count, which should equal the number of solutions in the container.

phcpy2c3.py2c_schubert_pieri_system()
Fills the container of systems with coefficients in standard double precision with a polynomial system that
expresses the intersection conditions of a general Pieri problem. On input are five integers and one string: 1)
m, the dimension of the input planes; 2) p, the dimension of the output planes; 3) q, the degree of the solution
maps; 4) nc, the number of characters in the string A; 5) A, m*p + q*(m+p) random complex input m-planes,
where the real and imaginary parts are separated by a space; 6) a flag is_real: if == 1, then the coefficients of A
are real, if == 0, then the coefficients of A are complex. Returns the failure code, which equals zero if all went
well.

phcpy2c3.py2c_schubert_quaddobl_littlewood_richardson_homotopies()
Runs the Littlewood-Richardson homotopies to resolve a number of general Schubert intersection conditions on
k-planes in n-space, in quad double precision. The polynomial system that was solved is in the container for
systems with coefficients in quad double precision and the corresponding solutions are in the quaddobl solutions
container. On entry are seven integers and two strings, in the following order: 1) n, the ambient dimension,
where the k-planes live; 2) k, the dimension of the solution planes; 3) c, the number of intersection conditions;
4) nc, the number of characters in the string brackets; 5) brackets is a string representation of c brackets, where
the numbers in each bracket are separated by spaces; 6) the flag verbose: when 0, no intermediate output is
written, when 1, then the resolution is dispayed on screen; 7) the flag tosquare: if 0, then no diagnostic verification is
done, if 1, then diagnostic verification is written to file; 8) the flag minrep: if 0, then all minors are used in the
system, if 1, then a minimal representation of the problem is used; 9) the flag tosquare: if 0, then Gauss-Newton
path trackers run, if 1, then the overdetermined systems are squared; 10) nbchar, the number of characters in
the string filename; 11) filename is the name of the output file. The function returns a tuple of an integer and a
string: 0) r is the formal root count as the number of k-planes for conditions imposed by the brackets for general
flags; 1) flags, a string with the coefficients of the general flags.

phcpy2c3.py2c_schubert_resolve_conditions()
Resolves a general Schubert intersection condition in n-space for k-planes subject to conditions defined by
brackers. On return is the root count, the number of k-planes that satisfy the intersection conditions imposed
by the brackets for general flags. On entry are five integers and one string: 1) n, the ambient dimension, where
the k-planes live; 2) k, the dimension of the solution planes; 3) c, the number of intersection conditions; 4) nc,
the number of characters in the string brackets; 5) brackets is a string representation of c brackets, where
the numbers in each bracket are separated by spaces; 6) the flag verbose: when 0, no intermediate output is written,
when 1, then the resolution is dispayed on screen.

phcpy2c3.py2c_schubert_standard_littlewood_richardson_homotopies()
Runs the Littlewood-Richardson homotopies to resolve a number of general Schubert intersection conditions on
k-planes in n-space, in standard double precision. The polynomial system that was solved is in the container for
systems with coefficients in standard double precision and the corresponding solutions are in the standard
solutions container. On entry are seven integers and two strings, in the following order: 1) n, the ambient
dimension, where the k-planes live; 2) k, the dimension of the solution planes; 3) c, the number of intersection conditions;
4) nc, the number of characters in the string brackets; 5) brackets is a string representation of c brackets, where
the numbers in each bracket are separated by spaces; 6) the flag verbose: if 0, then no intermediate output is written,
if 1, then the resolution is dispayed on screen; 7) the flag tosquare: if 0, then no diagnostic verification is done, if 1, then diagnostic verification is written to file; 8) the flag minrep: if 0, then all minors are used in the
system, if 1, then a minimal representation of the problem is used; 9) the flag tosquare: if 0, then Gauss-Newton
path trackers run, if 1, then the overdetermined systems are squared; 10) nbchar, the number of characters in
the string filename; 11) filename is the name of the output file. The function returns a tuple of an integer and a
string: 0) r is the formal root count as the number of k-planes for conditions imposed by the brackets for general flags; 1) flags, a string with the coefficients of the general flags.
phcpy2c3.py2c_set_gamma_constant()
Stores the gamma constant for later retrieval. Four parameters are expected on input, two doubles and two integers. The two doubles are the real and imaginary parts of the gamma. The two integers are the precision, 1, 2, or 4, respectively for double, double double, or quad double; and the verbose level.

phcpy2c3.py2c_set_seed()
Takes the value of the integer given on input and sets the seed for the random number generators. This fixing of the seed enables reproducible runs.

phcpy2c3.py2c_set_value_of_continuation_parameter()
Sets the value of a continuation parameter. On input is the index of this continuation parameter, an integer ranging from 1 to 34; and the new value for the continuation parameter. On return is a double with the value of the corresponding parameter.

phcpy2c3.py2c_show_continuation_parameters()
Shows the current values of the continuation parameters.

phcpy2c3.py2c_solcon_append_dobldobl_solution_string()
Appends a solution in double double precision to the list of solutions already stored in the container. There are three input parameters: 1) the number of variables; 2) the number of characters in the string; 3) the string representing the solution to append to the list. Returns the failure code, which equals zero if all went well.

phcpy2c3.py2c_solcon_append_multprec_solution_string()
Appends a solution in arbitrary multiprecision to the list of solutions already stored in the container. There are three input parameters: 1) the number of variables; 2) the number of characters in the string; 3) the string representing the solution to append to the list. Returns the failure code, which equals zero if all went well.

phcpy2c3.py2c_solcon_append_quaddobl_solution_string()
Appends a solution in quad double precision to the list of solutions already stored in the container. There are three input parameters: 1) the number of variables; 2) the number of characters in the string; 3) the string representing the solution to append to the list. Returns the failure code, which equals zero if all went well.

phcpy2c3.py2c_solcon_append_standard_solution_string()
Appends a solution in standard double precision to the list of solutions already stored in the container. There are three input parameters: 1) the number of variables; 2) the number of characters in the string; 3) the string representing the solution to append to the list. Returns the failure code, which equals zero if all went well.

phcpy2c3.py2c_solcon_clear_dobldobl_solutions()
Deallocates the container for solutions in double double precision. Returns the failure code, which equals zero when all went well.

phcpy2c3.py2c_solcon_clear_multprec_solutions()
Deallocates the container for solutions in arbitrary multiprecision. Returns the failure code, which equals zero when all went well.

phcpy2c3.py2c_solcon_clear_quaddobl_solutions()
Deallocates the container for solutions in quad double precision. Returns the failure code, which equals zero when all went well.

phcpy2c3.py2c_solcon_clear_standard_solutions()
Deallocates the container for solutions in standard double precision. Returns the failure code, which equals zero when all went well.

phcpy2c3.py2c_solcon_dobldobl_drop_coordinate_by_index()
Replaces the solutions in the double double precision container with the same solutions that have their k-th coordinate dropped. There is one input parameter: the index k of the coordinate. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_solcon_dobldobl_drop_coordinate_by_name()
Replaces the solutions in the double double precision container with the same solutions that have their coordinate
dropped corresponding to the name in the string \texttt{s} of \texttt{nc} characters long. There are two input parameters, an integer and a string: 1) \texttt{nc}, the number of characters in the string with the name; 2) \texttt{s}, the string with the name of the variable. On return is the failure code, which equals zero if all went well.

\begin{Verbatim}
\texttt{phcpy2c3.py2c_solcon_dobldobl_one_affinization}()
\end{Verbatim}

Divides every coordinate by the last coordinate of every solution in the container for solutions in double double precision.

\begin{Verbatim}
\texttt{phcpy2c3.py2c_solcon_dobldobl_one_homogenization}()
\end{Verbatim}

Add one extra coordinate one to every solution in the container for solutions in double double precision.

\begin{Verbatim}
\texttt{phcpy2c3.py2c_solcon_length_current_dobldobl_solution_string}()
\end{Verbatim}

Returns the number of characters in the string representation of the current double double solution in the container, at the place indicated by the value of the cursor. If this value equals zero, then there is no current solution, and then the length on return equals zero.

\begin{Verbatim}
\texttt{phcpy2c3.py2c_solcon_length_current_multprec_solution_string}()
\end{Verbatim}

Returns the number of characters in the string representation of the current arbitrary multiprecision solution in the container, at the place indicated by the value of the cursor. If this value equals zero, then there is no current solution, and then the length on return equals zero.

\begin{Verbatim}
\texttt{phcpy2c3.py2c_solcon_length_current_quaddobl_solution_string}()
\end{Verbatim}

Returns the number of characters in the string representation of the current quad double solution in the container, at the place indicated by the value of the cursor. If this value equals zero, then there is no current solution, and then the length on return equals zero.

\begin{Verbatim}
\texttt{phcpy2c3.py2c_solcon_length_current_standard_solution_string}()
\end{Verbatim}

Returns the number of characters in the string representation of the current standard double solution in the container, at the place indicated by the value of the cursor. If this value equals zero, then there is no current solution, and then the length on return equals zero.

\begin{Verbatim}
\texttt{phcpy2c3.py2c_solcon_length_dobldobl_solution_string}()
\end{Verbatim}

On input is the index \texttt{k} to a solution in double double precision, stored in the container. On return is the length of the string representation for that \texttt{k}-th solution in the container.

\begin{Verbatim}
\texttt{phcpy2c3.py2c_solcon_length_multprec_solution_string}()
\end{Verbatim}

On input is the index \texttt{k} to a solution in arbitrary multiprecision, stored in the container. On return is the length of the string representation for that \texttt{k}-th solution in the container.

\begin{Verbatim}
\texttt{phcpy2c3.py2c_solcon_length_quaddobl_solution_string}()
\end{Verbatim}

On input is the index \texttt{k} to a solution in quad double precision, stored in the container. On return is the length of the string representation for that \texttt{k}-th solution in the container.

\begin{Verbatim}
\texttt{phcpy2c3.py2c_solcon_length_standard_solution_string}()
\end{Verbatim}

On input is the index \texttt{k} to a solution in standard double precision, stored in the container. On return is the length of the string representation for that \texttt{k}-th solution in the container.

\begin{Verbatim}
\texttt{phcpy2c3.py2c_solcon_move_current_dobldobl_to_next}()
\end{Verbatim}

Moves the pointer to the current solution in double double precision to the next solution and returns the value of the cursor. If cursor on return is zero, then either the pointer was null or there is no next solution.

\begin{Verbatim}
\texttt{phcpy2c3.py2c_solcon_move_current_multprec_to_next}()
\end{Verbatim}

Moves the pointer to the current solution in arbitrary multiprecision to the next solution and returns the value of the cursor. If cursor on return is zero, then either the pointer was null or there is no next solution.

\begin{Verbatim}
\texttt{phcpy2c3.py2c_solcon_move_current_quaddobl_to_next}()
\end{Verbatim}

Moves the pointer to the current solution in quad double precision to the next solution and returns the value of the cursor. If cursor on return is zero, then either the pointer was null or there is no next solution.
phcpy2c3.py2c_solcon_move_current_standard_to_next()
Moves the pointer to the current solution in standard double precision to the next solution and returns the value of the cursor. If cursor on return is zero, then either the pointer was null or there is no next solution.

phcpy2c3.py2c_solcon_number_of_dobldobl_solutions()
Returns the number of solutions in double double precision, as stored in the container.

phcpy2c3.py2c_solcon_number_of_multprec_solutions()
Returns the number of solutions in arbitrary multiprecision, as stored in the container.

phcpy2c3.py2c_solcon_number_of_quaddobl_solutions()
Returns the number of solutions in quad double precision, as stored in the container.

phcpy2c3.py2c_solcon_number_of_standard_solutions()
Returns the number of solutions in standard double precision, as stored in the container.

phcpy2c3.py2c_solcon_open_solution_input_file()
Prompts the user for the name of the input file for the solutions and opens the input file. All subsequent reading happens from this input. Returns the failure code, which equals zero when all went well.

phcpy2c3.py2c_solcon_quaddobl_drop_coordinate_by_index()
Replaces the solutions in the quad double precision container with the same solutions that have their k-th coordinate dropped. There is one input parameter: the index k of the coordinate. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_solcon_quaddobl_drop_coordinate_by_name()
Replaces the solutions in the quad double precision container with the same solutions that have their coordinate dropped corresponding to the name in the string s of nc characters long. There are two input parameters, an integer and a string: 1) nc, the number of characters in the string with the name; 2) s, the string with the name of the variable. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_solcon_quaddobl_one_affinization()
Divides every coordinate by the last coordinate of every solution in the container for solutions in quad double precision.

phcpy2c3.py2c_solcon_quaddobl_one_homogenization()
Add one extra coordinate one to every solution in the container for solutions in double double precision.

phcpy2c3.py2c_solcon_read_dobldobl_solutions()
Interactive function to read the solutions into the container, in double double precision. Returns the failure code, which is zero when all went well.

phcpy2c3.py2c_solcon_read_dobldobl_solutions_from_file()
The two input arguments are a number and a string: 1) The number equals the number of characters in the string. 2) The string given on input is the name of a file which contains a solution list to be parsed in double double precision. Solutions are read from file and stored in the container for double double precision solutions. The failure code is returned, which is zero if all went well.

phcpy2c3.py2c_solcon_read_multprec_solutions()
Interactive function to read the solutions into the container, in arbitrary multiprecision. Returns the failure code, which is zero when all went well.

phcpy2c3.py2c_solcon_read_quaddobl_solutions()
Interactive function to read the solutions into the container, in quad double precision. Returns the failure code, which is zero when all went well.

phcpy2c3.py2c_solcon_read_quaddobl_solutions_from_file()
The two input arguments are a number and a string: 1) The number equals the number of characters in the string. 2) The string given on input is the name of a file which contains a solution list to be parsed in quad double precision. Solutions are read from file and stored in the container for quad double precision solutions. The failure code is returned, which is zero if all went well.
phcpy2c3.py2c_solcon_read_standard_solutions()
Interactive function to read the solutions into the container, in standard double precision. Returns the failure code, which is zero when all went well.

phcpy2c3.py2c_solcon_read_standard_solutions_from_file()
The two input arguments are a number and a string: 1) The number equals the number of characters in the string. 2) The string given on input is the name of a file which contains a solution list to be parsed in standard double precision. Solutions are read from file and stored in the container for double precision solutions. The failure code is returned, which is zero if all went well.

phcpy2c3.py2c_solcon_retrieve_next_dobldobl_initialize()
Resets the pointer to the current dobldobl solution in the container to the first solution in the list. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_solcon_retrieve_next_multprec_initialize()
Resets the pointer to the current multprec solution in the container to the first solution in the list. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_solcon_retrieve_next_quaddobl_initialize()
Resets the pointer to the current quaddobl solution in the container to the first solution in the list. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_solcon_retrieve_next_standard_initialize()
Resets the pointer to the current standard solution in the container to the first solution in the list. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_solcon_standard_drop_coordinate_by_index()
Replaces the solutions in the standard double precision container with the same solutions that have their k-th coordinate dropped. There is one input parameter: the index k of the coordinate. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_solcon_standard_drop_coordinate_by_name()
Replaces the solutions in the standard double precision container with the same solutions that have their coordinate dropped corresponding to the name in the string s of nc characters long. There are two input parameters, an integer and a string: 1) nc, the number of characters in the string with the name; 2) s, the string with the name of the variable. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_solcon_standard_one_affinization()
Divides every coordinate by the last coordinate of every solution in the container for solutions in standard double precision.

phcpy2c3.py2c_solcon_standard_one_homogenization()
Add one extra coordinate one to every solution in the container for solutions in standard double precision.

phcpy2c3.py2c_solcon_write_current_dobldobl_solution_string()
Writes the current double double solution in the solution container to the string s of n+1 characters. The last character is the end of string symbol. The value of n is given as the one input parameter to this function. On return is the string that contains the string representation of the current solution in standard double precision in the container.

phcpy2c3.py2c_solcon_write_current_multprec_solution_string()
Writes the current arbitrary multiprecision solution in the solution container to the string s of n+1 characters. The last character is the end of string symbol. The value of n is given as the one input parameter to this function. On return is the string that contains the string representation of the current solution in standard double precision in the container.

phcpy2c3.py2c_solcon_write_current_quaddobl_solution_string()
Writes the current quad double solution in the solution container to the string s of n+1 characters. The last character is the end of string symbol. The value of n is given as the one input parameter to this function. On
return is the string that contains the string representation of the current solution in standard double precision in the container.

phcpy2c3.py2c_solcon_write_current_standard_solution_string()
Writes the current standard double solution in the solution container to the string s of n+1 characters. The last character is the end of string symbol. The value of n is given as the one input parameter to this function. On return is the string that contains the string representation of the current solution in standard double precision in the container.

phcpy2c3.py2c_solcon_write_dobldobl_solution_string()
Returns the string representation for the k-th solution stored in double double precision in the container. On input are two integers: 1) the index to the solution; and 2) the number of characters in the string representation for that solution.

phcpy2c3.py2c_solcon_write_dobldobl_solutions()
Writes the solutions in double double precision to screen. Returns the failure code, which equals zero when all went well.

phcpy2c3.py2c_solcon_write_multprec_solution_string()
Returns the string representation for the k-th solution stored in arbitrary multiprecision in the container. On input are two integers: 1) the index to the solution; and 2) the number of characters in the string representation for that solution.

phcpy2c3.py2c_solcon_write_multprec_solutions()
Writes the solutions in arbitrary multiprecision to screen. Returns the failure code, which equals zero when all went well.

phcpy2c3.py2c_solcon_write_quaddobl_solution_string()
Returns the string representation for the k-th solution stored in quad double precision in the container. On input are two integers: 1) the index to the solution; and 2) the number of characters in the string representation for that solution.

phcpy2c3.py2c_solcon_write_quaddobl_solutions()
Writes the solutions in quad double precision to screen. Returns the failure code, which equals zero when all went well.

phcpy2c3.py2c_solcon_write_standard_solution_string()
Returns the string representation for the k-th solution stored in standard double precision in the container. On input are two integers: 1) the index to the solution; and 2) the number of characters in the string representation for that solution.

phcpy2c3.py2c_solcon_write_standard_solutions()
Writes the solutions in standard double precision to screen. Returns the failure code, which equals zero when all went well.

phcpy2c3.py2c_solve_by_dobldobl_Laurent_homotopy_continuation()
Tracks the paths defined by the homotopy in double double precision to solve a Laurent system stored in the systems container, starting at the solutions of a stored Laurent start system. On input is one integer: the number of tasks for path tracking. If that input number is zero, then no multitasking is applied. On return is the failure code, which is zero when all went well.

phcpy2c3.py2c_solve_by_dobldobl_homotopy_continuation()
Tracks the paths defined by the homotopy in double double precision. On input is one integer: the number of tasks for path tracking. If that input number is zero, then no multitasking is applied. On return is the failure code, which is zero when all went well.

phcpy2c3.py2c_solve_by_multprec_homotopy_continuation()
Tracks the paths defined by the homotopy in arbitrary multiprecision. On input is one integer: the number of decimal places in the precision. On return is the failure code, which is zero when all went well.
Tracks the paths defined by the homotopy in quad double precision to solve a Laurent system stored in the systems container, starting at the solutions of a stored Laurent start system. On input is one integer: the number of tasks for path tracking. If that input number is zero, then no multitasking is applied. On return is the failure code, which is zero when all went well.

Tracks the paths defined by the homotopy in quad double precision. On input is one integer: the number of tasks for path tracking. If that input number is zero, then no multitasking is applied. On return is the failure code, which is zero when all went well.

Tracks the paths defined by the homotopy in standard double precision to solve a Laurent system stored in the systems container, starting at the solutions of a stored Laurent start system. On input is one integer: the number of tasks for path tracking. If that input number is zero, then no multitasking is applied. On return is the failure code, which is zero when all went well.

Tracks the paths defined by the homotopy in standard double precision. On input is one integer: the number of tasks for path tracking. If that input number is zero, then no multitasking is applied. On return is the failure code, which is zero when all went well.

Calls the blackbox solver on the system stored in the container for Laurent systems with coefficients in double double precision. Three integers are expected on input: 1) a boolean flag silent: if 1, then no intermediate output about the root counts is printed, if 0, then the solver is verbose; 2) the number of tasks: if 0, then no multitasking is applied, otherwise as many tasks as the number will run; and 3) the verbose level. On return, the container for solutions in double double precision contains the solutions to the system in the double double Laurent systems container.

Calls the blackbox solver on the system stored in the container for systems with coefficients in double double precision. Three integers are expected on input: 1) a boolean flag if silent: if 1, then no intermediate output about the root counts is printed, if 0, then the solver is verbose; 2) the number of tasks. If that number is zero, then no multitasking is applied; and 3) the verbose level. On return, the container for solutions in double double precision contains the solutions to the system in the dobldobl systems container.

Calls the blackbox solver on the system stored in the container for Laurent systems with coefficients in quad double precision. Three integers are expected on input: 1) a boolean flag silent: if 1, then no intermediate output about the root counts is printed, if 0, then the solver is verbose; 2) the number of tasks: if 0, then no multitasking is applied, otherwise as many tasks as the number will run; and 3) the verbose level. On return, the container for solutions in quad double precision contains the solutions to the system in the quad double Laurent systems container.

Calls the blackbox solver on the system stored in the container for systems with coefficients in quad double precision. Three integers are expected on input: 1) a boolean flag silent: if 1, then no intermediate output about the root counts is printed, if 0, then the solver is verbose; 2) the number of tasks. If that number is zero, then no multitasking is applied; and 3) the verbose level. On return, the container for solutions in quad double precision contains the solutions to the system in the quaddouble systems container.

Calls the blackbox solver on the system stored in the container for Laurent systems with coefficients in standard double precision. Four integers are expected on input: 1) a boolean flag silent: if 1, then no intermediate output about the root counts is printed, if 0, then the solver is verbose; 2) the number of tasks: if 0, then no multitasking is applied, otherwise as many tasks as the number will run; 3) a flag to focus on mixed volumes and polyhedral homotopies, if 1, the no degree bounds will be computed, as when the system is genuinely Laurent with negative
exponents; and 4) the verbose level. On return, the container for solutions in standard double precision contains the solutions to the system in the standard Laurent systems container.

\texttt{phcpy2c3.py2c\_solve\_standard\_system}()

Calls the blackbox solver on the system stored in the container for systems with coefficients in standard double precision. Four integers are expected on input: 1) a boolean flag if silent: if 1, then no intermediate output about the root counts is printed, if 0, then the solver is verbose; 2) the number of tasks. If that number is zero, then no multitasking is applied; 3) a flag to focus on mixed volumes and polyhedral homotopies, if 1, then no degree bounds will be computed; and 4) the verbose level. On return, the container for solutions in standard double precision contains the solutions to the system in the standard systems container.

\texttt{phcpy2c3.py2c\_standard\_Laurent\_cascade\_homotopy}()

Creates a homotopy in standard double precision using the stored Laurent systems to go one level down the cascade, removing one slice. On return is the failure code, which equals zero if all went well.

\texttt{phcpy2c3.py2c\_standard\_Newton\_Laurent\_step}()

Applies one Newton step in standard double precision to the Laurent system in the standard Laurent systems container and to the solutions in the container. On return is the failure code, which equals zero if all went well.

\texttt{phcpy2c3.py2c\_standard\_Newton\_power\_series}()

Given in the systems container a polynomial system with coefficients in standard double precision, and in the standard systems pool the leading terms of the power series, this function runs Newton’s method to compute power series solutions of the system in the container, in standard double precision. There are four integers on input: 1) the index of the series parameter; 2) the maximal degree of the series; 3) the number of Newton steps to be done on each solution; 4) a 0/1-flag to indicate whether additional diagnostic output needs to be written to screen. On return is the failure code, which equals zero if all went well.

\texttt{phcpy2c3.py2c\_standard\_Newton\_series}()

Given in the systems container a polynomial system with coefficients in standard double precision, and in the solutions container the leading coefficients of the power series, this function runs Newton’s method to compute power series solutions of the system in the container, in standard double precision. There are four integers on input: 1) the index of the series parameter; 2) the maximal degree of the series; 3) the number of Newton steps to be done on each solution; 4) a 0/1-flag to indicate whether additional diagnostic output needs to be written to screen. On return is the failure code, which equals zero if all went well.

\texttt{phcpy2c3.py2c\_standard\_Newton\_step}()

Applies one Newton step in standard double precision to the system in the standard systems container and to the solutions in the container. On return is the failure code, which equals zero if all went well.

\texttt{phcpy2c3.py2c\_standard\_Pade\_approximant}()

Given in the systems container a polynomial system with coefficients in standard double precision, and in the solutions container the leading coefficients of the power series, this function runs Newton’s method to compute power series solutions of the system in the container, in standard double precision, followed by the construction of the Pade approximants, for each solution. There are five integers on input: 1) the index of the series parameter; 2) the degree of the numerator of the Pade approximant; 3) the degree of the denominator of the Pade approximant; 4) the number of Newton steps to be done on each solution; 5) a 0/1-flag to indicate whether additional diagnostic output needs to be written to screen. The Pade approximants are stored in the standard systems pool, numerators in the odd indexed entries and denominators in the entries with even index in each system. On return is the failure code, which equals zero if all went well.

\texttt{phcpy2c3.py2c\_standard\_cascade\_homotopy}()

Creates a homotopy in standard double precision using the stored systems to go one level down the cascade, removing one slice. On return is the failure code, which equals zero if all went well.

\texttt{phcpy2c3.py2c\_standard\_collapse\_diagonal}()

Eliminates the extrinsic diagonal for the system and solutions in the containers for standard doubles. On input are two integers: 1) k, the current number of slack variables in the embedding; 2) d, the number of slack variables to add to the final embedding. The system in the container has its diagonal eliminated and is embedded with
k+d slack variables. The solutions corresponding to this system are in the solutions container. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_standard_condition_report()**
For the system and solutions in the containers in double precision, computes a condition report. On input are the following: 1) maximum number of Newton iterations per solution; 2) tolerance on the residual; 3) tolerance on the forward error; 4) tolerance on the inverse condition number for singularities; 5) a string with the name of the output file, this string may be empty if no output to file is needed; 6) a verbose flag, either 1 or 0. On return are the counts of number of solutions that are regular, singular, real, complex, clustered, or failures; along with the frequency tables for the forward errors, residuals and estimates for the inverse condition numbers.

**phcpy2c3.py2c_standard_crude_tracker()**
A crude tracker appends the end point of a path directly to the solutions container, without refinement or post-processing. Tracking happens in standard double precision. On entry is the verbose parameter which is 1 or 0. If 1, then the solution vectors are written to screen, otherwise the crude tracker stays mute. On return is the failure code, which is zero when all went well. The requirement is that the target system, start system, and start solutions in standard double precision have been initialized in the containers.

**phcpy2c3.py2c_standard_deflate()**
Applies deflation in standard double precision to the system and the solutions stored in the containers. There are five input parameters, two integers and three doubles: (1) maxitr : the maximum number of iterations per root, (2) maxdef : the maximum number of deflations per root, (3) toler : tolerance on the forward error on each root, (4) tolres : tolerance on the backward error on each root, (5) toles : tolerance on the numerical rank of the Jacobian matrices. On return is the failure code, which equals zero if all went well.

**phcpy2c3.py2c_standard_diagonal_cascade_solutions()**
Makes the start solutions to start the cascade homotopy to intersect two solution sets of dimensions a and b, where a >= b, in standard double precision. The dimensions a and b are given as input parameters. The systems stored as target and start system in the container define the witness sets for these two solution sets. On return is the failure code, which equals zero when all went well.

**phcpy2c3.py2c_standard_diagonal_homotopy()**
Creates a diagonal homotopy to intersect two solution sets of dimensions a and b respectively, where a >= b. The two input parameters are values for a and b. The systems stored as target and start system in the container, in standard double precision, define the witness sets for these two solution sets.

**phcpy2c3.py2c_standard_laursys_solve()**
Runs the cascades of homotopies on the Laurent polynomial system in the standard systems container. Runs in standard double precision. On input are five integers : 1) nbtasks equals the number of tasks for multitasking, 2) topdim is the top dimension to start the homotopy cascades, 3) filter is a 0 or 1 flag to filter the witness supersets, 4) factor is a 0 or 1 flag to factor the witness sets, 5) verbose is a flag for intermediate output.

**phcpy2c3.py2c_standard_multiplicity_structure()**
Computes the multiplicity structure in standard double precision. Required is the presence of a polynomial system in the standard systems container and a solution in the standard solutions container. The input parameters are two integers and one double: order : the maximum differentiation order, verbose : 1 for verbose, 0 for silent, and tol : tolerance on the numerical rank. On return is a tuple: the multiplicity and the values of the Hilbert function.

**phcpy2c3.py2c_standard_polysys_solve()**
Runs the cascades of homotopies on the polynomial system in the standard systems container. Runs in standard double precision. On input are five integers : 1) nbtasks equals the number of tasks for multitasking, 2) topdim is the top dimension to start the homotopy cascades, 3) filter is a 0 or 1 flag to filter the witness supersets, 4) factor is a 0 or 1 flag to factor the witness sets, 5) verbose is a flag for intermediate output.

**phcpy2c3.py2c_standard_witset_of_Laurent_hypersurface()**
Given in the string p of nc characters a polynomial in nv variables, terminated by a semicolon, the systems and solutions container in standard double precision on return contain a witness set for the hypersurface defined by
the Laurent polynomial in p. On entry are two integers and one string, in the following order: 1) nv, the number
of variables of the polynomials; 2) nc, the number of characters in the string p; 3) p, string representation of
a Laurent polynomial in several variables, terminates with a semicolon. On return is the failure code, which
equals zero if all went well.

`phcpy2c3.py2c_standard_witset_of_hypersurface()`
Given in the string p of nc characters a polynomial in nv variables, terminated by a semicolon, the systems and
solutions container in standard double precision on return contain a witness set for the hypersurface defined by
the ordinary polynomial in p. On entry are two integers and one string, in the following order: 1) nv, the number
of variables of the polynomials; 2) nc, the number of characters in the string p; 3) p, string representation of
an ordinary polynomial in several variables, terminates with a semicolon. On return is the failure code, which
equals zero if all went well.

`phcpy2c3.py2c_swap_symbols_for_dobldobl_Laurent_witness_set()`
Permutes the slack variables in the Laurent system with double double precision coefficients and its corre-
sponding solutions in the containers so the slack variables appear at the end. On input are two integers: the
total number of variables; and the number of slack variables, or the dimension of the set. This permutation is
necessary to consider the system and solutions stored in containers as a witness set.

`phcpy2c3.py2c_swap_symbols_for_dobldobl_witness_set()`
Permutes the slack variables in the polynomial system with double double precision coefficients and its corre-
sponding solutions in the containers so the slack variables appear at the end. On input are two integers: the
total number of variables; and the number of slack variables, or the dimension of the set. This permutation is
necessary to consider the system and solutions stored in containers as a witness set.

`phcpy2c3.py2c_swap_symbols_for_quaddobl_Laurent_witness_set()`
Permutes the slack variables in the Laurent system with quad double precision coefficients and its corresponding
solutions in the containers so the slack variables appear at the end. On input are two integers: the total number
of variables; and the number of slack variables, or the dimension of the set. This permutation is necessary to
consider the system and solutions stored in containers as a witness set.

`phcpy2c3.py2c_swap_symbols_for_quaddobl_witness_set()`
Permutes the slack variables in the polynomial system with quad double precision coefficients and its corresponding
solutions in the containers so the slack variables appear at the end. On input are two integers: the total number
of variables; and the number of slack variables, or the dimension of the set. This permutation is necessary to
consider the system and solutions stored in containers as a witness set.

`phcpy2c3.py2c_swap_symbols_for_standard_Laurent_witness_set()`
Permutes the slack variables in the Laurent system with standard double precision coefficients and its corresponding
solutions in the containers so the slack variables appear at the end. On input are two integers: the total number
of variables; and the number of slack variables, or the dimension of the set. This permutation is necessary to
consider the system and solutions stored in containers as a witness set.

`phcpy2c3.py2c_swap_symbols_for_standard_witness_set()`
Permutes the slack variables in the polynomial system with standard double precision coefficients and its corresponding
solutions in the containers so the slack variables appear at the end. On input are two integers: the total number
of variables; and the number of slack variables, or the dimension of the set. This permutation is necessary to
consider the system and solutions stored in containers as a witness set.

`phcpy2c3.py2c_sweep_clear_definitions()`
Clears the definitions in the sweep homotopy.

`phcpy2c3.py2c_sweep_define_parameters_numerically()`
Defines the indices to the variables that serve as parameters numerically, that is: via integer indices. On entry
are three integer numbers and a string. The string is a string representation of a Python list of integers, The three
integers are the number of equations, the number of variables, and the number of parameters. The number of
variables m includes the number of parameters. Then there should be as many as m indices in the list of integers
to define which of the variables are parameters.
phcpy2c3.py2c_sweep_define_parameters_symbolically()
Defines the indices to the variables that serve as parameters symbolically, that is, as names of variables. For this to work, the symbol table must be initialized. On entry are four integer numbers and a string. The four integers are the number of equations, the number of variables, the number of parameters (the number of variables m includes the number of parameters), and the number of characters in the string. The string contains the names of the parameters, separated by one comma. For this to work, the symbol table must be initialized, e.g.: via the reading of a polynomial system.

phcpy2c3.py2c_sweep_dobldobl_complex_run()
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. Moreover, the containers of systems and solutions in double double precision have been initialized with a parametric systems and start solutions. The first input parameter 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 given in 2 pointers to doubles.

phcpy2c3.py2c_sweep_dobldobl_real_run()
There are no input arguments to this routine. 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]. This family must be stored in the systems container in double double precision and the corresponding start solutions in the dobldobl solutions container, 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.

phcpy2c3.py2c_sweep_get_dobldobl_start()
Gets the start values for the parameters in double double precision, giving on input the number n of doubles that need to be returned. On return will be n doubles, for the consecutive real and imaginary parts for the start values of all parameters, stored in the string representation of a Python list of doubles.

phcpy2c3.py2c_sweep_get_dobldobl_target()
Gets the target values for the parameters in double double precision, giving on input the number n of doubles that need to be returned. On return will be n doubles, for the consecutive real and imaginary parts for the target values of all parameters, stored in the string representation of a Python list of doubles.

phcpy2c3.py2c_sweep_get_indices_numerically()
Returns the indices of the variables that are parameters, as the string representation of a Python list of integers.

phcpy2c3.py2c_sweep_get_indices_symbolically()
Returns a string with the names of the parameters, each separated by one space.

phcpy2c3.py2c_sweep_get_number_of_equations()
Returns the number of equations in the sweep homotopy.

phcpy2c3.py2c_sweep_get_number_of_parameters()
Returns the number of parameters in the sweep homotopy.

phcpy2c3.py2c_sweep_get_number_of_variables()
Returns the number of variables in the sweep homotopy.

phcpy2c3.py2c_sweep_get_quaddobl_start()
Gets the start values for the parameters in quad double precision, giving on input the number n of doubles that need to be returned. On return will be n doubles, for the consecutive real and imaginary parts for the start values of all parameters, stored in the string representation of a Python list of doubles.

phcpy2c3.py2c_sweep_get_quaddobl_target()
Returns the target values for the parameters in quad double precision, giving on input the number n of doubles that need to be returned. On return will be n doubles, for the consecutive real and imaginary parts for the target values of all parameters, stored in the string representation of a Python list of doubles.
phcpy2c3.\texttt{py2c\_sweep\_get\_standard\_start}()  
Gets the start values for the parameters in standard double precision, giving on input the number \( n \) of doubles that need to be returned. On return will be \( n \) doubles, for the consecutive real and imaginary parts for the start values of all parameters, stored in the string representation of a Python list of doubles.

phcpy2c3.\texttt{py2c\_sweep\_get\_standard\_target}()  
Gets the target values for the parameters in standard double precision, giving on input the number \( n \) of doubles that need to be returned. On return will be \( n \) doubles, for the consecutive real and imaginary parts for the target values of all parameters, stored in the string representation of a Python list of doubles.

phcpy2c3.\texttt{py2c\_sweep\_quaddobl\_complex\_run}()  
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. Moreover, the containers of systems and solutions in quad double precision have been initialized with a parametric systems and start solutions. The first input parameter 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 given in 2 pointers to doubles.

phcpy2c3.\texttt{py2c\_sweep\_quaddobl\_real\_run}()  
There are no input arguments to this routine. 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 \((1-s)\ast(A - v[0]) + s\ast(A - v[1])\), where \( A \) is the natural parameter, going from the start value \( v[0] \) to the target value \( v[1] \). This family must be stored in the systems container in quad double precision and the corresponding start solutions in the quaddobl solutions container, 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.

phcpy2c3.\texttt{py2c\_sweep\_set\_dobldobl\_start}()  
Sets the start values for the \( m \) parameters in double double precision, giving on input an integer \( m \) and \( 4\ast m \) doubles, with the consecutive real and imaginary parts for the start values of all \( m \) parameters.

phcpy2c3.\texttt{py2c\_sweep\_set\_dobldobl\_target}()  
Sets the target values for the \( m \) parameters in double double precision, giving on input an integer \( m \) and \( 4\ast m \) doubles, with the consecutive real and imaginary parts for the target values of all \( m \) parameters.

phcpy2c3.\texttt{py2c\_sweep\_set\_quaddobl\_start}()  
Sets the start values for the \( m \) parameters in quad double precision, giving on input an integer \( m \) and \( 8\ast m \) doubles, with the consecutive real and imaginary parts for the start values of all \( m \) parameters.

phcpy2c3.\texttt{py2c\_sweep\_set\_quaddobl\_target}()  
Sets the target values for the \( m \) parameters in quad double precision, giving on input an integer \( m \) and \( 8\ast m \) doubles, with the consecutive real and imaginary parts for the target values of all \( m \) parameters.

phcpy2c3.\texttt{py2c\_sweep\_set\_standard\_start}()  
Sets the start values for the \( m \) parameters in standard double precision, giving on input an integer \( m \) and \( 2\ast m \) doubles, with the consecutive real and imaginary parts for the start values of all \( m \) parameters. The doubles are given in a string representation of a Python list of doubles.

phcpy2c3.\texttt{py2c\_sweep\_set\_standard\_target}()  
Sets the target values for the \( m \) parameters in standard double precision, giving on input an integer \( m \) and \( 2\ast m \) doubles, with the consecutive real and imaginary parts for the target values of all \( m \) parameters.

phcpy2c3.\texttt{py2c\_sweep\_standard\_complex\_run}()  
Starts the trackers in a complex convex parameter homotopy, in standard double precision, where the indices to the parameters, start and target values are already defined. Moreover, the containers of systems and solutions in standard double precision have been initialized with a parametric systems and start solutions. The first input parameter 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 given in 2 pointers to doubles.

phcpy2c3.\texttt{py2c\_sweep\_standard\_real\_run}()
There are no input arguments to this routine. 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]. This family must be stored in the systems container in standard double precision and the corresponding start solutions in the standard solutions container, 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.

**phcpy2c3.py2c_syscon_add_symbol()**

Adds a symbol to the table, with name given in the string, where the number of characters in the name equals the first integer argument. The second input parameter is the string. This symbol represents the last variable added in the homogeneous coordinate transformation.

**phcpy2c3.py2c_syscon_clear_dobldobl_Laurent_system()**

Deallocates the container for Laurent polynomial systems with coefficients in double double precision.

**phcpy2c3.py2c_syscon_clear_dobldobl_system()**

Deallocates the container for polynomial systems with coefficients in double double precision.

**phcpy2c3.py2c_syscon_clear_multprec_Laurent_system()**

Deallocates the container for Laurent polynomial systems with coefficients in arbitrary multiprecision.

**phcpy2c3.py2c_syscon_clear_multprec_system()**

Deallocates the container for polynomial systems with coefficients in arbitrary multiprecision.

**phcpy2c3.py2c_syscon_clear_quaddobl_Laurent_system()**

Deallocates the container for Laurent polynomial systems with coefficients in quad double precision.

**phcpy2c3.py2c_syscon_clear_quaddobl_system()**

Deallocates the container for polynomial systems with coefficients in quad double precision.

**phcpy2c3.py2c_syscon_clear_standard_Laurent_system()**

Deallocates the container for Laurent polynomial systems with coefficients in standard double precision.

**phcpy2c3.py2c_syscon_clear_standard_system()**

Deallocates the container for polynomial systems with coefficients in standard double precision.

**phcpy2c3.py2c_syscon_clear_symbol_table()**

Clears the symbol table.

**phcpy2c3.py2c_syscon_degree_of_dobldobl_polynomial()**

Returns the degree of the k-th polynomial in the container for polynomials with coefficients in double double precision. The index k of the polynomial is the one input argument.

**phcpy2c3.py2c_syscon_degree_of_multprec_polynomial()**

Returns the degree of the k-th polynomial in the container for polynomials with coefficients in arbitrary multiprecision. The index k of the polynomial is the one input argument.

**phcpy2c3.py2c_syscon_degree_of_quaddobl_polynomial()**

Returns the degree of the k-th polynomial in the container for polynomials with coefficients in quad double precision. The index k of the polynomial is the one input argument.

**phcpy2c3.py2c_syscon_degree_of_standard_polynomial()**

Returns the degree of the k-th polynomial in the container for polynomials with coefficients in standard double precision. The index k of the polynomial is the one input argument.

**phcpy2c3.py2c_syscon_dobldobl_Laurent_drop_variable_by_index()**

Replaces the Laurent system in the double double precision container with the same Laurent system that has its k-th variable dropped. The index k of the variable is given as an input parameter. On return is the failure code, which equals zero if all went well.

4.8. the module phcpy.phcpy2c3

199
Replaces the Laurent system in the double double precision container with the same Laurent system that have that variable dropped corresponding to the name in the string s of nc characters long. The function has two input parameters, an integer and a string: 1) nc, the number of characters in the string with the name; 2) s, a string that holds the name of the variable. On return is the failure code, which equals zero if all went well.

Replaces the system in the double double precision container with the same system that has its k-th variable dropped. The index k of the variable is given as an input parameter. On return is the failure code, which equals zero if all went well.

Replaces the system in the double double precision container with the same system that have that variable dropped corresponding to the name in the string s of nc characters long. The function has two input parameters, an integer and a string: 1) nc, the number of characters in the string with the name; 2) s, a string that holds the name of the variable. On return is the failure code, which equals zero if all went well.

Replaces the system in the double double precision container with its transformation to affine coordinates, substituting the value of the last coordinate by one and removing the last equation.

Replaces the system in the double double precision container with its transformation in 1-homogeneous coordinates. There is one integer on input. If 0, then a random linear equation is added, otherwise, the linear equation $z_0 - 1 = 0$ is added, where $z_0$ is the extra homogeneous coordinate.

Places in the systems container a random polynomial system with coefficients in double double precision. There are five integers as input parameters: 1) n, the number of polynomials and variables; 2) m, the number of monomials per equation; 3) d, the largest degree of each monomial; 4) c, the type of coefficient: 0 if on the complex unit circle, 1, if all coefficients are one, 2, if all coefficients are random floats in [-1,+1]; 5) neq, the number of polynomials in the system.

Initializes the container for Laurent polynomials with coefficients in double double precision. The input argument is an integer, the number of polynomials in the container. The failure code is returned, which equals zero if all went well.

Initializes the container for polynomials with coefficients in double double precision. The input argument is an integer, the number of polynomials in the container. The failure code is returned, which equals zero if all went well.

Initializes the container for Laurent polynomials with coefficients in arbitrary multiprecision. The input argument is an integer, the number of polynomials in the container. The failure code is returned, which equals zero if all went well.

Initializes the container for polynomials with coefficients in arbitrary multiprecision. The input argument is an integer, the number of polynomials in the container. The failure code is returned, which equals zero if all went well.

Initializes the container for Laurent polynomials with coefficients in quad double precision. The input argument is an integer, the number of polynomials in the container. The failure code is returned, which equals zero if all went well.
Initializes the container for polynomials with coefficients in quad double precision. The input argument is an integer, the number of polynomials in the container. The failure code is returned, which equals zero if all went well.

phcpy2c3.py2c_syscon_initialize_number_of_standard_Laurentians()

Initializes the container for Laurent polynomials with coefficients in standard double precision. The input argument is an integer, the number of polynomials in the container. The failure code is returned, which equals zero if all went well.

phcpy2c3.py2c_syscon_initialize_number_of_standard_polynomials()

Initializes the container for polynomials with coefficients in standard double precision. The input argument is an integer, the number of polynomials in the container. The failure code is returned, which equals zero if all went well.

phcpy2c3.py2c_syscon_load_dobldobl_Laurential()

Returns the k-th polynomial in the Laurent systems container with double double complex coefficients as a string. The value for k is in the one integer parameter of this function.

phcpy2c3.py2c_syscon_load_dobldobl_polynomial()

Returns the k-th polynomial in the systems container with double double complex coefficients as a string. The value for k is in the one integer parameter of this function.

phcpy2c3.py2c_syscon_load_multprec_Laurential()

Returns the k-th polynomial in the Laurent systems container with arbitrary multiprecision complex coefficients as a string. The value for k is in the one integer parameter of this function.

phcpy2c3.py2c_syscon_load_multprec_polynomial()

Returns the k-th polynomial in the systems container with arbitrary multiprecision complex coefficients as a string. The value for k is in the one integer parameter of this function.

phcpy2c3.py2c_syscon_load_quaddobl_Laurential()

Returns the k-th polynomial in the Laurent systems container with quad double complex coefficients as a string. The value for k is in the one integer parameter of this function.

phcpy2c3.py2c_syscon_load_quaddobl_polynomial()

Returns the k-th polynomial in the systems container with quad double complex coefficients as a string. The value for k is in the one integer parameter of this function.

phcpy2c3.py2c_syscon_load_standard_Laurential()

Returns the k-th polynomial in the Laurent systems container with standard double complex coefficients as a string. The value for k is in the one integer parameter of this function.

phcpy2c3.py2c_syscon_load_standard_polynomial()

Returns the k-th polynomial in the systems container with standard double complex coefficients as a string. The value for k is in the one integer parameter of this function.

phcpy2c3.py2c_syscon_number_of_Laurent_terms()

Returns the number of terms in the k-th Laurent polynomial stored in the container for Laurent polynomials systems with coefficients in standard double precision. The input parameter k is the index of the polynomial k.

phcpy2c3.py2c_syscon_number_of_dobldobl_Laurentials()

Returns the number of Laurent polynomials with coefficients in double double precision as stored in the systems container.

phcpy2c3.py2c_syscon_number_of_dobldobl_polynomials()

Returns the number of polynomials with coefficients in double double precision as stored in the systems container.

phcpy2c3.py2c_syscon_number_of_multprec_Laurentials()

Returns the number of Laurent polynomials with coefficients in arbitrary multiprecision as stored in the systems container.
Returns the number of polynomials with coefficients in arbitrary multiprecision as stored in the systems container.

Returns the number of Laurent polynomials with coefficients in quad double precision as stored in the systems container.

Returns the number of polynomials with coefficients in quad double precision as stored in the systems container.

Returns the number of Laurent polynomials with coefficients in standard double precision as stored in the systems container.

Returns the number of polynomials with coefficients in standard double precision as stored in the systems container.

Returns the number of symbols in the symbol table.

Returns the number of terms in the k-th polynomial stored in the container for systems with coefficients in standard double precision. The input parameter k is the index of the polynomial k.

Replaces the Laurent system in the quad double precision container with the same Laurent system that has its k-th variable dropped. The index k of the variable is given as an input parameter. On return is the failure code, which equals zero if all went well.

Replaces the Laurent system in the quad double precision container with the same Laurent system that have that variable dropped corresponding to the name in the string s of nc characters long. The function has two input parameters, an integer and a string: 1) nc, the number of characters in the string with the name; 2) s, a string that holds the name of the variable. On return is the failure code, which equals zero if all went well.

Replaces the system in the quad double precision container with the same system that has its k-th variable dropped. The index k of the variable is given as an input parameter. On return is the failure code, which equals zero if all went well.

Replaces the system in the quad double precision container with the same system that have that variable dropped corresponding to the name in the string s of nc characters long. The function has two input parameters, an integer and a string: 1) nc, the number of characters in the string with the name; 2) s, a string that holds the name of the variable. On return is the failure code, which equals zero if all went well.

Replaces the system in the quad double precision container by its transformation to affine coordinates, substituting the value of the last coordinate by one and removing the last equation.

Replaces the system in the quad double precision container with its transformation in 1-homogeneous coordinates. There is one integer on input. If 0, then a random linear equation is added, otherwise, the linear equation $z_0 - 1 = 0$ is added, where $z_0$ is the extra homogeneous coordinate.

Places in the systems container a random polynomial system with coefficients in quad double precision. There are five integers as input parameters: 1) n, the number of polynomials and variables; 2) m, the number of
monomials per equation; 3) \(d\), the largest degree of each monomial; 4) \(c\), the type of coefficient: 0 if on the complex unit circle, 1, if all coefficients are one, 2, if all coefficients are random floats in \([-1,+1]\); 5) \(neq\), the number of polynomials in the system.

\[\text{phcpy2c3.py2c_syscon_random_system}\]

Places in the systems container a random polynomial system with coefficients in standard double precision. There are five integers as input parameters: 1) \(n\), the number of polynomials and variables; 2) \(m\), the number of monomials per equation; 3) \(d\), the largest degree of each monomial; 4) \(c\), the type of coefficient: 0 if on the complex unit circle, 1, if all coefficients are one, 2, if all coefficients are random floats in \([-1,+1]\); 5) \(neq\), the number of polynomials in the system.

\[\text{phcpy2c3.py2c_syscon_read_dobldobl_Laurent_system}\]

Interactive procedure to read a Laurent polynomial system with coefficients in double double precision. The system will be placed in the dobldobl Laurent systems container. The failure code is returned, which equals zero if all went well.

\[\text{phcpy2c3.py2c_syscon_read_dobldobl_system}\]

Interactive procedure to read a polynomial system with coefficients in double double precision. The system will be placed in the dobldobl systems container. The failure code is returned, which equals zero if all went well.

\[\text{phcpy2c3.py2c_syscon_read_multprec_Laurent_system}\]

Interactive procedure to read a Laurent polynomial system with coefficients in arbitrary multiprecision. The one input parameter is an integer, the number of decimal places in the working precision. The system will be placed in the multprec Laurent systems container. The failure code is returned, which equals zero if all went well.

\[\text{phcpy2c3.py2c_syscon_read_multprec_system}\]

Interactive procedure to read a polynomial system with coefficients in arbitrary multiprecision. The one input parameter is an integer, the number of decimal places in the working precision. The system will be placed in the multprec systems container. The failure code is returned, which equals zero if all went well.

\[\text{phcpy2c3.py2c_syscon_read_quaddobl_Laurent_system}\]

Interactive procedure to read a Laurent polynomial system with coefficients in quad double precision. The system will be placed in the quaddobl Laurent systems container. The failure code is returned, which equals zero if all went well.

\[\text{phcpy2c3.py2c_syscon_read_quaddobl_system}\]

Interactive procedure to read a polynomial system with coefficients in quad double precision. The system will be placed in the quaddobl systems container. The failure code is returned, which equals zero if all went well.

\[\text{phcpy2c3.py2c_syscon_read_standard_Laurent_system}\]

Interactive procedure to read a Laurent polynomial system with coefficients in standard double precision. The system will be placed in the standard Laurent systems container. The failure code is returned, which equals zero if all went well.

\[\text{phcpy2c3.py2c_syscon_read_standard_system}\]

Interactive procedure to read a polynomial system with coefficients in standard double precision. The system will be placed in the standard systems container. The failure code is returned, which equals zero if all went well.

\[\text{phcpy2c3.py2c_syscon_remove_symbol_name}\]

Removes a symbol, given by name, from the symbol table. On input are two arguments: 1) an integer, as the number of characters in the name; 2) a string of characters with the name of the symbol. The failure code is returned, which equals zero when all went well.

\[\text{phcpy2c3.py2c_syscon_retrieve_term}\]

Retrieves one term of a polynomial with coefficients in standard double precision, that is stored in the systems container. On return is the failure code, which equals zero if all went well.

\[\text{phcpy2c3.py2c_syscon_standard_Laurent_drop_variable_by_index}\]

Replaces the Laurent system in the standard double precision container with the same Laurent system that has
its k-th variable dropped. The index k of the variable is given as an input parameter. On return is the failure code, which equals zero if all went well.

\texttt{phcpy2c3.py2c_syscon_standard_Laurent_drop_variable_by_name()}

Replaces the Laurent system in the standard double precision container with the same Laurent system that have that variable dropped corresponding to the name in the string s of nc characters long. The function has two input parameters, an integer and a string: 1) nc, the number of characters in the string with the name; 2) s, a string that holds the name of the variable. On return is the failure code, which equals zero if all went well.

\texttt{phcpy2c3.py2c_syscon_standard_drop_variable_by_index()}

Replaces the system in the standard double precision container with the same system that has its k-th variable dropped. The index k of the variable is given as an input parameter. On return is the failure code, which equals zero if all went well.

\texttt{phcpy2c3.py2c_syscon_standard_drop_variable_by_name()}

Replaces the system in the standard double precision container with the same system that have that variable dropped corresponding to the name in the string s of nc characters long. The function has two input parameters, an integer and a string: 1) nc, the number of characters in the string with the name; 2) s, a string that holds the name of the variable. On return is the failure code, which equals zero if all went well.

\texttt{phcpy2c3.py2c_syscon_standard_one_affinization()}

Replaces the system in the standard double precision container by its transformation to affine coordinates, substituting the value of the last coordinate by one and removing the last equation.

\texttt{phcpy2c3.py2c_syscon_standard_one_homogenization()}

Replaces the system in the standard double precision container with its transformation in 1-homogeneous coordinates. There is one integer on input. If 0, then a random linear equation is added, otherwise, the linear equation \( z_0 - 1 = 0 \) is added, where \( z_0 \) is the extra homogeneous coordinate.

\texttt{phcpy2c3.py2c_syscon_store_dobldobl_Laurential()}

Defines the k-th polynomial in the systems container for Laurent polynomials with coefficients in double double precision. As a precondition for this function, the container must be initialized for sufficiently many polynomials, in any case \( \geq k \). There are four input parameters, three integers and one string: 1) nc, the number of characters in the string p; 2) n, the number of variables in the multivariate polynomial; 3) k, the index of the polynomial in the system; 4) p, a valid string representation for a polynomial. On return is the failure code, which equals zero when all went well.

\texttt{phcpy2c3.py2c_syscon_store_dobldobl_polynomial()}

Defines the k-th polynomial in the systems container for polynomials with coefficients in double double precision. As a precondition for this function, the container must be initialized for sufficiently many polynomials, in any case \( \geq k \). There are four input parameters, three integers and one string: 1) nc, the number of characters in the string p; 2) n, the number of variables in the multivariate polynomial; 3) k, the index of the polynomial in the system; 4) p, a valid string representation for a polynomial. On return is the failure code, which equals zero when all went well.

\texttt{phcpy2c3.py2c_syscon_store_multprec_Laurential()}

Defines the k-th polynomial in the systems container for Laurent polynomials with coefficients in arbitrary multiprecision. As a precondition for this function, the container must be initialized for sufficiently many polynomials, in any case \( \geq k \). There are five input parameters, four integers and one string: 1) nc, the number of characters in the string p; 2) n, the number of variables in the multivariate polynomial; 3) k, the index of the polynomial in the system; 4) dp, the number of decimal places to parse the coefficients; 5) p, a valid string representation for a polynomial. On return is the failure code, which equals zero when all went well.

\texttt{phcpy2c3.py2c_syscon_store_multprec_polynomial()}

Defines the k-th polynomial in the systems container for polynomials with coefficients in arbitrary multiprecision. As a precondition for this function, the container must be initialized for sufficiently many polynomials, in any case \( \geq k \). There are five input parameters, four integers and one string: 1) nc, the number of characters in the string p; 2) n, the number of variables in the multivariate polynomial; 3) k, the index of the polynomial in the system; 4) p, a valid string representation for a polynomial. On return is the failure code, which equals zero when all went well.
the system; 4) dp, the number of decimal places to parse the coefficients; 5) p, a valid string representation for a polynomial. On return is the failure code, which equals zero when all went well.

phcpy2c3.py2c_syscon_store_quaddobl_Laurential()

Defines the k-th polynomial in the systems container for Laurent polynomials with coefficients in quad double precision. As a precondition for this function, the container must be initialized for sufficiently many polynomials, in any case $\geq k$. There are four input parameters, three integers and one string: 1) nc, the number of characters in the string p; 2) n, the number of variables in the multivariate polynomial; 3) k, the index of the polynomial in the system; 4) p, a valid string representation for a polynomial. On return is the failure code, which equals zero when all went well.

phcpy2c3.py2c_syscon_store_quaddobl_polynomial()

Defines the k-th polynomial in the systems container for polynomials with coefficients in quad double precision. As a precondition for this function, the container must be initialized for sufficiently many polynomials, in any case $\geq k$. There are four input parameters, three integers and one string: 1) nc, the number of characters in the string p; 2) n, the number of variables in the multivariate polynomial; 3) k, the index of the polynomial in the system; 4) p, a valid string representation for a polynomial. On return is the failure code, which equals zero when all went well.

phcpy2c3.py2c_syscon_store_standard_Laurential()

Defines the k-th polynomial in the systems container for Laurent polynomials with coefficients in standard double precision. As a precondition for this function, the container must be initialized for sufficiently many polynomials, in any case $\geq k$. There are four input parameters, three integers and one string: 1) nc, the number of characters in the string p; 2) n, the number of variables in the multivariate polynomial; 3) k, the index of the polynomial in the system; 4) p, a valid string representation for a polynomial. On return is the failure code, which equals zero when all went well.

phcpy2c3.py2c_syscon_store_standard_polynomial()

 Defines the k-th polynomial in the systems container for polynomials with coefficients in standard double precision. As a precondition for this function, the container must be initialized for sufficiently many polynomials, in any case $\geq k$. There are four input parameters, three integers and one string: 1) nc, the number of characters in the string p; 2) n, the number of variables in the multivariate polynomial; 3) k, the index of the polynomial in the system; 4) p, a valid string representation for a polynomial. On return is the failure code, which equals zero when all went well.

phcpy2c3.py2c_syscon_string_of_symbols()

Returns a string that contains the symbols in the symbol table. The symbols are separate from each other by one space.

phcpy2c3.py2c_syscon_total_degree()

Returns in d the total degree of the system with coefficients in standard double precision, as stored in the container.

phcpy2c3.py2c_syscon_write_dobldobl_Laurent_system()

Writes the Laurent polynomial system with double double precision coefficients that is stored in the container.

phcpy2c3.py2c_syscon_write_dobldobl_system()

Writes the polynomial system with double double precision coefficients that is stored in the container.

phcpy2c3.py2c_syscon_write_multprec_Laurent_system()

Writes the Laurent polynomial system with arbitrary multiprecision coefficients that is stored in the container.

phcpy2c3.py2c_syscon_write_multprec_system()

Writes the polynomial system with arbitrary multiprecision coefficients that is stored in the container.

phcpy2c3.py2c_syscon_write_quaddobl_Laurent_system()

Writes the Laurent polynomial system with quad double precision coefficients that is stored in the container.

phcpy2c3.py2c_syscon_write_quaddobl_system()

Writes the polynomial system with quad double precision coefficients that is stored in the container.
phcpy2c3.py2c_syscon_write_standard_Laurent_system()
    Writes the Laurent polynomial system with standard double precision coefficients that is stored in the container.

phcpy2c3.py2c_syscon_write_standard_system()
    Writes the polynomial system with standard double precision coefficients that is stored in the container.

phcpy2c3.py2c_syscon_write_symbols()
    Writes the symbols in the symbol table to screen. Returns the failure code, which equals zero if all went well.

phcpy2c3.py2c_syspool_copy_to_dobldobl_container()
    Copies the k-th system in the pool for systems in double double precision to the dobldobl systems container. The value for k is given as an integer input parameter. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_syspool_copy_to_quaddobl_container()
    Copies the k-th system in the pool for systems in quad double precision to the quaddobl systems container. The value for k is given as an integer input parameter. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_syspool_copy_to_standard_container()
    Copies the k-th system in the pool for systems in standard double precision to the standard systems container. The value for k is given as an integer input parameter. On return is the failure code, which equals zero if all went well.

phcpy2c3.py2c_syspool_dobldobl_clear()
    Clears the pool for systems in double double precision.

phcpy2c3.py2c_syspool_dobldobl_create()
    Defines the k-th system in the dobldobl system pool, using the system in the dobldobl container.

phcpy2c3.py2c_syspool_dobldobl_init()
    Initializes the pool for systems in double double precision.

phcpy2c3.py2c_syspool_dobldobl_size()
    Returns the size of the pool for systems in double double precision.

phcpy2c3.py2c_syspool_quaddobl_clear()
    Clears the pool for systems in quad double precision.

phcpy2c3.py2c_syspool_quaddobl_create()
    Defines the k-th system in the quaddobl system pool, using the system in the quaddobl container.

phcpy2c3.py2c_syspool_quaddobl_init()
    Initializes the pool for systems in quad double precision.

phcpy2c3.py2c_syspool_quaddobl_size()
    Returns the size of the pool for systems in quad double precision.

phcpy2c3.py2c_syspool_standard_clear()
    Clears the pool for systems in standard double precision.

phcpy2c3.py2c_syspool_standard_create()
    Defines the k-th system in the standard system pool, using the system in the standard container.

phcpy2c3.py2c_syspool_standard_init()
    Initializes the pool for systems in standard double precision.

phcpy2c3.py2c_syspool_standard_size()
    Returns the size of the pool for systems in standard double precision.

phcpy2c3.py2c_tabform_load_standard_tableau()
    Returns a 5-tuple with the tableau form of the system with standard double precision coefficients in the container. On input is the verbose flag, as an integer. The five items in the returned tuple are 1) the number of equations
as an integer, 2) the number of equations as an integer, 3) the number of terms in each polynomial, given as a string, the string representation of a list of integers, 4) the coefficients of all terms, given as a string, the string representation of a list of doubles, each pair of consecutive doubles represents a complex coefficient, 5) the exponents of all terms, given as a string, the string representation of a list of integers.

\texttt{phcpy2c3.py2c\_tabform\_store\_standard\_tableau()}

On input is the tableau form of a polynomial system, given by 1) the number of equations as an integer, 2) the number of equations as an integer, 3) the number of characters in the 4-th string input, 4) the number of terms in each polynomial, given as a string, the string representation of a list of integers, 5) the number of characters in the 6-th string input, 6) the coefficients of all terms, given as a string, the string representation of a list of doubles, each pair of consecutive doubles represents a complex coefficient, 7) the number of characters in the 7-th string input, 8) the exponents of all terms, given as a string, the string representation of a list of integers. The tableau form is parsed and the container for systems with standard double precision coefficients is initialized.

\texttt{phcpy2c3.py2c\_tune\_continuation\_parameters()}

Interactive procedure to tune the continuation parameters.

\texttt{phcpy2c3.py2c\_usolve\_dobldobl()}

Applies the method of Weierstrass to compute all roots of a polynomial in one variable with double double precision arithmetic. On input are two numbers: 1) the maximum number of iterations in the method of Weierstrass; and 2) the epsilon requirement on the accuracy of the roots. Before calling this function, the polynomial should be stored in the dobldobl systems container. After the call of this function, the dobldobl solutions container contains the roots of the polynomial. On return is the number of iterations done by the solver.

\texttt{phcpy2c3.py2c\_usolve\_multprec()}

Applies the method of Weierstrass to compute all roots of a polynomial in one variable with arbitrary multiprecision arithmetic. On input are three numbers: 1) the number of decimal places in the working precision; 2) the maximum number of iterations in the method of Weierstrass; and 3) the epsilon requirement on the accuracy of the roots. Before calling this function, the polynomial should be stored in the multprec systems container. After the call of this function, the multprec solutions container contains the roots of the polynomial. On return is the number of iterations done by the solver.

\texttt{phcpy2c3.py2c\_usolve\_quaddobl()}

Applies the method of Weierstrass to compute all roots of a polynomial in one variable with quad double precision arithmetic. On input are two numbers: 1) the maximum number of iterations in the method of Weierstrass; and 2) the epsilon requirement on the accuracy of the roots. Before calling this function, the polynomial should be stored in the quaddobl systems container. After the call of this function, the quaddobl solutions container contains the roots of the polynomial. On return is the number of iterations done by the solver.

\texttt{phcpy2c3.py2c\_usolve\_standard()}

Applies the method of Weierstrass to compute all roots of a polynomial in one variable with standard double precision arithmetic. On input are two numbers: 1) the maximum number of iterations in the method of Weierstrass; and 2) the epsilon requirement on the accuracy of the roots. Before calling this function, the polynomial should be stored in the standard systems container. After the call of this function, the standard solutions container contains the roots of the polynomial. On return is the number of iterations done by the solver.

\texttt{phcpy2c3.py2c\_varbprec\_Newton\_Laurent\_steps()}

Applies Newton’s method in variable precision. There are six input parameters: 1) the dimension: the number of variables and equations; 2) the accuracy, expressed as the correct number of decimal places; 3) the maximum number of iterations in Newton’s method; 4) an upper bound on the number of decimal places in the precision; 5) a string, with the representation of the polynomials in the system. On return is the failure code, which equals zero if all went well.

\texttt{phcpy2c3.py2c\_witset\_dobldobl\_Laurent\_ismember()}

Runs the homotopy membership test for a point to belong to a witness set defined by a Laurent polynomial system in double double precision, where the test point is given as a string in PHCpack format. The containers in double double precision must contain the embedded Laurent system and the corresponding generic points. On entry are seven parameters. The first four are integers: 1) vrbb, an integer flag (0 or 1) for the verbosity...
of the test, 2) nvr, the ambient dimension, number of coordinates of the point, 3) dim, the dimension of the witness set, 4) nbc, the number of characters in the string representing the point, the test point is represented as a solution string in symbolic format, including the symbols for the variables, before the coordinates; the next two parameters are two doubles; 5) restol, tolerance on the residual for the valuation of the point, 6) homtol, tolerance on the homotopy membership test for the point; and the last parameter is a string: 7) tpt, the string representation of a solution which contains the coordinates of the test point in symbolic format. On return are three 0/1 integers, to be interpreted as booleans: 1) fail, the failure code of the procedure, 2) onsys, 0 if the evaluation test failed, 1 if success, 3) onset, 0 if not a member of the witness set, 1 if a member.

```python
phcpy2c3.py2c_witset_dobldobl_Laurent_membertest()
```

Executes the homotopy membership test for a point to belong to a witness set defined by a Laurent polynomial system in double double precision. The containers in double double precision must contain the embedded Laurent system and its corresponding solutions for the witness set of a positive dimensional solution set. On entry are the seven parameters, the first four are integers: 1) vrb, an integer flag (0 or 1) for the verbosity of the test, 2) nvr, the ambient dimension, number of coordinates of the point, 3) dim, the dimension of the witness set, 4) nbc, the number of characters in the string representing the point; the next two parameters are two doubles: 5) restol, tolerance on the residual for the valuation of the point, 6) homtol, tolerance on the homotopy membership test for the point; and the last parameter is a string: 7) tpt, the string representation of a solution which contains the coordinates of the test point in symbolic format. On return are three 0/1 integers, to be interpreted as booleans: 1) fail, the failure code of the procedure, 2) onsys, 0 if the evaluation test failed, 1 if success, 3) onset, 0 if not a member of the witness set, 1 if a member.

```python
phcpy2c3.py2c_witset_dobldobl_ismember()
```

Runs the homotopy membership test for a point to belong to a witness set defined by an ordinary polynomial system in double double precision, where the test point is given as a string in PHCpack format. The containers in double double precision must contain the embedded system and the corresponding generic points. On entry are the seven parameters. The first four are integers: 1) vrb, an integer flag (0 or 1) for the verbosity of the test, 2) nvr, the ambient dimension, number of coordinates of the point, 3) dim, the dimension of the witness set, 4) nbc, the number of characters in the string representing the point, the test point is represented as a solution string in symbolic format, including the symbols for the variables, before the coordinates; the next two parameters are two doubles: 5) restol, tolerance on the residual for the valuation of the point, 6) homtol, tolerance on the homotopy membership test for the point; and the last parameter is a string: 7) tpt, the string representation of the point as a list with as many as 4*nvr doubles for the real and imaginary parts of the double double precision coordinates of the test point. On return are three 0/1 integers, to be interpreted as booleans: 1) fail, the failure code of the procedure, 2) onsys, 0 if the evaluation test failed, 1 if success, 3) onset, 0 if not a member of the witness set, 1 if a member.

```python
phcpy2c3.py2c_witset_quaddobl_Laurent_ismember()
```

Runs the homotopy membership test for a point to belong to a witness set defined by a Laurent polynomial system in quad double precision, where the test point is given as a string in PHCpack format. The containers in quad double precision must contain the embedded Laurent system and the corresponding generic points. On
entry are seven parameters. The first four are integers: 1) vr
, an integer flag (0 or 1) for the verbosity of the test,
2) nvr, the ambient dimension, number of coordinates of the point, 3) dim, the dimension of the witness set, 4) nb
, the number of characters in the string representing the point, the test point is represented as a solution string
in symbolic format, including the symbols for the variables, before the coordinates; the next two parameters are
two doubles: 5) restol, tolerance on the residual for the valuation of the point, 6) homtol, tolerance on the
homotopy membership test for the point; and the last parameter is a string: 7) tpt, the string representation of a
solution which contains the coordinates of the test point in symbolic format. On return are three 0/1 integers, to
be interpreted as booleans: 1) fail, the failure code of the procedure, 2) onsys, 0 if the evaluation test failed, 1 if
success, 3) onset, 0 if not a member of the witness set, 1 if a member.

phcpy2c3.py2c_witset_quaddobl_Laurent_membertest ()
Executes the homotopy membership test for a point to belong to a witness set defined by a Laurent polynomial
system in quad double precision. The containers in quad double precision must contain the embedded Laurent
system and its corresponding solutions for the witness set of a positive dimensional solution set. On entry are
the seven parameters, the first four are integers: 1) vr
, an integer flag (0 or 1) for the verbosity of the test, 2) n
, the ambient dimension, number of coordinates of the point, 3) dim, the dimension of the witness set, 4) nb
, the number of characters in the string representing the point, the next two parameters are two doubles: 5) restol,
tolerance on the residual for the valuation of the point, 6) homtol, tolerance on the homotopy membership test
for the point; and the last parameter is a string: 7) tpt, the string representation of a solution containing the coordinates of the test point in symbolic format. On return are three 0/1 integers, to be interpreted as booleans: 1) fail, the failure code of the procedure, 2) onsys, 0 if the evaluation test failed, 1 if success, 3) onset, 0 if not a member of the witness set, 1 if a member.

phcpy2c3.py2c_witset_quaddobl_ismember ()
Runs the homotopy membership test for a point to belong to a witness set defined by an ordinary polynomial
system in quad double precision, where the test point is given as a string in PHCpack format. The containers in
quad double precision must contain the embedded system and the corresponding generic points. On entry are
seven parameters. The first four are integers: 1) vr
, an integer flag (0 or 1) for the verbosity of the test, 2) n
, the ambient dimension, number of coordinates of the point, 3) dim, the dimension of the witness set, 4) nb
, the number of characters in the string representing the point, the next two parameters are two doubles: 5) restol,
tolerance on the residual for the valuation of the point, 6) homtol, tolerance on the homotopy membership test
for the point; and the last parameter is a string: 7) tpt, the string representation of a solution containing the coordinates of the test point in symbolic format. On return are three 0/1 integers, to be interpreted as booleans: 1) fail, the failure code of the procedure, 2) onsys, 0 if the evaluation test failed, 1 if success, 3) onset, 0 if not a member of the witness set, 1 if a member.

phcpy2c3.py2c_witset_quaddobl_membertest ()
Executes the homotopy membership test for a point to belong to a witness set defined by an ordinary polynomial
system in quad double precision. The containers in quad double precision must contain the embedded polynomial
system and its corresponding solutions for the witness set of a positive dimensional solution set. On entry are
the seven parameters, the first four are integers: 1) vr
, an integer flag (0 or 1) for the verbosity of the test, 2) n
, the ambient dimension, number of coordinates of the point, 3) dim, the dimension of the witness set, 4) nb
, the number of characters in the string representing the point, the next two parameters are two doubles: 5) restol,
tolerance on the residual for the valuation of the point, 6) homtol, tolerance on the homotopy membership test
for the point; and the last parameter is a string: 7) tpt, the string representation of a solution containing the coordinates of the test point in symbolic format. On return are three 0/1 integers, to be interpreted as booleans: 1) fail, the failure code of the procedure, 2) onsys, 0 if the evaluation test failed, 1 if success, 3) onset, 0 if not a member of the witness set, 1 if a member.

phcpy2c3.py2c_witset_standard_Laurent_ismember ()
Runs the homotopy membership test for a point to belong to a witness set defined by a Laurent polynomial
system in standard double precision, where the test point is given as a string in PHCpack format. The containers in
standard double precision must contain the embedded Laurent system and the corresponding generic points. On entry are seven parameters. The first four are integers: 1) vr
, an integer flag (0 or 1) for the verbosity of the test, 2) n
, the ambient dimension, number of coordinates of the point, 3) dim, the dimension of the
witness set, 4) nbc, the number of characters in the string representing the point, the test point is represented as a solution string in symbolic format, including the symbols for the variables, before the coordinates; the next two parameters are two doubles: 5) restol, tolerance on the residual for the valuation of the point, 6) homtol, tolerance on the homotopy membership test for the point; and the last parameter is a string: 7) tpt, the string representation of a solution which contains the coordinates of the test point in symbolic format. On return are three 0/1 integers, to be interpreted as booleans: 1) fail, the failure code of the procedure, 2) onsys, 0 if the evaluation test failed, 1 if success, 3) onset, 0 if not a member of the witness set, 1 if a member.

\texttt{phcpy2c3.py2c_witset_standard_Laurent_membertest()}

Executes the homotopy membership test for a point to belong to a witness set defined by a Laurent polynomial system in standard double precision. The containers in standard double precision must contain the embedded Laurent system and its corresponding solutions for the witness set of a positive dimensional solution set. On entry are the seven parameters, the first four are integers: 1) vrb, an integer flag (0 or 1) for the verbosity of the test, 2) nvr, the ambient dimension, number of coordinates of the point, 3) dim, the dimension of the witness set, 4) nbc, the number of characters in the string representing the point, the next two parameters are two doubles: 5) restol, tolerance on the residual for the valuation of the point, 6) homtol, tolerance on the homotopy membership test for the point; and the last parameter is a string: 7) tpt, the string representation of the point as a list with as many as $2 \times \text{nvr}$ doubles for the real and imaginary parts of the standard double precision coordinates of the test point. On return are three 0/1 integers, to be interpreted as booleans: 1) fail, the failure code of the procedure, 2) onsys, 0 if the evaluation test failed, 1 if success, 3) onset, 0 if not a member of the witness set, 1 if a member.

\texttt{phcpy2c3.py2c_witset_standard_ismember()}

Runs the homotopy membership test for a point to belong to a witness set defined by an ordinary polynomial system in standard double precision, where the test point is given as a string in PHCpack format. The containers in standard double precision must contain the embedded system and the corresponding generic points. On entry are seven parameters. The first four are integers: 1) vrb, an integer flag (0 or 1) for the verbosity of the test, 2) nvr, the ambient dimension, number of coordinates of the point, 3) dim, the dimension of the witness set, 4) nbc, the number of characters in the string representing the point, the test point is represented as a solution string in symbolic format, including the symbols for the variables, before the coordinates; the next two parameters are two doubles: 5) restol, tolerance on the residual for the valuation of the point, 6) homtol, tolerance on the homotopy membership test for the point; and the last parameter is a string: 7) tpt, the string representation of a solution which contains the coordinates of the test point in symbolic format. On return are three 0/1 integers, to be interpreted as booleans: 1) fail, the failure code of the procedure, 2) onsys, 0 if the evaluation test failed, 1 if success, 3) onset, 0 if not a member of the witness set, 1 if a member.

\texttt{phcpy2c3.py2c_witset_standard_membertest()}

Executes the homotopy membership test for a point to belong to a witness set defined by an ordinary polynomial system in standard double precision. The containers in standard double precision must contain the embedded polynomial system and its corresponding solutions for the witness set of a positive dimensional solution set. On entry are the seven parameters, the first four are integers: 1) vrb, an integer flag (0 or 1) for the verbosity of the test, 2) nvr, the ambient dimension, number of coordinates of the point, 3) dim, the dimension of the witness set, 4) nbc, the number of characters in the string representing the point, the next two parameters are two doubles: 5) restol, tolerance on the residual for the valuation of the point, 6) homtol, tolerance on the homotopy membership test for the point; and the last parameter is a string: 7) tpt, the string representation of the point as a list with as many as $2 \times \text{nvr}$ doubles for the real and imaginary parts of the standard double precision coordinates of the test point. On return are three 0/1 integers, to be interpreted as booleans: 1) fail, the failure code of the procedure, 2) onsys, 0 if the evaluation test failed, 1 if success, 3) onset, 0 if not a member of the witness set, 1 if a member.

\texttt{phcpy2c3.py2c_write_dobldobl_start_Laurent_system()}

Writes the start Laurent system in double double precision.

\texttt{phcpy2c3.py2c_write_dobldobl_start_system()}

Writes the start system as stored in double double precision to screen or to the defined output file.

\texttt{phcpy2c3.py2c_write_dobldobl_target_Laurent_system()}

Writes the target Laurent system in double double precision.
phcpy2c3.py2c_write_dobldobl_target_system()
 Writes the target system as stored in double double precision to screen or to the defined output file.

phcpy2c3.py2c_write_quaddobl_start_Laurent_system()
 Writes the start Laurent system in quad double precision.

phcpy2c3.py2c_write_quaddobl_start_system()
 Writes the start system as stored in quad double precision to screen or to the defined output file.

phcpy2c3.py2c_write_quaddobl_target_Laurent_system()
 Writes the target Laurent system in quad double precision.

phcpy2c3.py2c_write_quaddobl_target_system()
 Writes the target system as stored in quad double precision to screen or to the defined output file.

phcpy2c3.py2c_write_standard_start_Laurent_system()
 Writes the start Laurent system in standard double precision.

phcpy2c3.py2c_write_standard_start_system()
 Writes the start system as stored in standard double precision to screen or to the defined output file.

phcpy2c3.py2c_write_standard_target_Laurent_system()
 Writes the target Laurent system in standard double precision.

phcpy2c3.py2c_write_standard_target_system()
 Writes the target system as stored in standard double precision to screen or to the defined output file.

phcpy2c3.py2c_write_start_solutions()
 Writes the start solutions in standard double precision either to the screen (standard output) or to the defined output file. On return is the failure code, which is zero if all is well.
CHAPTER 5

Indices and tables

- genindex
- modindex
- search
Python Module Index

cascades, 123
curves, 142
dashboards, 149
diagonal, 128
examples, 130
factor, 126
families, 132
interface, 150
maps, 138
phcpy2c3, 154
polynomials, 101
polytopes, 136
schubert, 134
series, 139
server, 71
sets, 117
solutions, 98
solver, 89
sweepers, 106
trackers, 101
tropisms, 114
tuning, 108
A
ade_double_double_track() (in module trackers), 101
ade_double_track() (in module trackers), 102
ade_quad_double_track() (in module trackers), 102
ade_tuned_double_double_track() (in module trackers), 102
ade_tuned_double_track() (in module trackers), 102
ade_tuned_quad_double_track() (in module trackers), 102
adjacent_minors() (in module families), 132
apollonius() (in module series), 139
B
binomials() (in module examples), 130
C
cascade_filter() (in module cascades), 123
cascade_step() (in module cascades), 123
cascades
 module, 123
chandra() (in module families), 132
cheater() (in module schubert), 134
check_mixture() (in module polytopes), 136
checkin_newton_power_series() (in module series), 139
corrector_abscorendgame_get() (in module tuning), 108
corrector_abscoronpath_get() (in module tuning), 108
corrector_abscoronpath_set() (in module tuning), 108
corrector_absresendgame_get() (in module tuning), 108
corrector_absresendgame_set() (in module tuning), 108
corrector_absresonpath_get() (in module tuning), 108
corrector_absresonpath_set() (in module tuning), 108
corrector_maxiterendgame_get() (in module tuning), 108
corrector_maxiterendgame_set() (in module tuning), 108
corrector_maxiteronpath_get() (in module tuning), 108
corrector_maxiteronpath_set() (in module tuning), 108
corrector_relcorendgame_get() (in module tuning), 109
corrector_relcorendgame_set() (in module tuning), 109
corrector_relcoronpath_get() (in module tuning), 109
corrector_relcoronpath_set() (in module tuning), 109
corrector_relresendgame_get() (in module tuning), 109
corrector_relresendgame_set() (in module tuning), 109
corrector_relresonpath_get() (in module tuning), 109
corrector_relresonpath_set() (in module tuning), 109
coordinates() (in module solutions), 98
corrector_abscorendgame_set() (solutions.Solution method), 98
corrector_abscoronpath_set() (in module tuning), 108
corrector_absresendgame_set() (in module tuning), 108
corrector_absresenpath_set() (in module tuning), 108
CoordinatePlot (class in dashboard), 149
curves
module, 142
cyclic() (in module families), 132
cyclic3homotopy() (in module trackers), 102
cyclic7() (in module examples), 130

d
dashboard
module, 149
decompose() (in module factor), 126
decomposition() (in module factor), 126
default_path_parameters() (in module tuning), 109
diagnostics() (in module solutions), 98
diagnostics() (solutions.Solution method), 98
diagonal
module, 128
diagonal_solver() (in module diagonal), 128
dictionary() (solutions.Solution method), 98
distance_to_endgame_get() (in module tuning), 109
distance_to_endgame_set() (in module tuning), 109
dobldobl_clear() (in module tropisms), 114
dobldobl_closest_pole() (in module curves), 142
dobldobl_complex_sweep() (in module sweepers), 107
dobldobl_decomposition() (in module factor), 126
dobldobl_deflate() (in module solver), 89
dobldobl_diagonal_cascade_solutions() (in module diagonal), 128
dobldobl_diagonal_homotopy() (in module diagonal), 128
dobldobl_diagonal_solver() (in module diagonal), 128
dobldobl_dimension() (in module tropisms), 114
dobldobl_embed() (in module sets), 117
dobldobl_estimated_distance() (in module curves), 142
dobldobl_get_solution() (in module curves), 142
dobldobl_hessian_step() (in module curves), 142
dobldobl_initialize() (in module tropisms), 114
dobldobl_ismember() (in module sets), 117
dobldobl_ismember_filter() (in module sets), 117
dobldobl_laurent_embed() (in module sets), 117
dobldobl_laurent_ismember() (in module sets), 117
dobldobl_laurent_ismember_filter() (in module sets), 117
dobldobl_laurent_membretest() (in module sets), 118
dobldobl_laursys_solve() (in module factor), 126
dobldobl_linear_reduction() (in module solver), 90
dobldobl_littlewood_richardson_homotopies() (in module schubert), 134
dobldobl_membretest() (in module sets), 118
dobldobl_monodromy_breakup() (in module factor), 126
dobldoblMultiplicity() (in module solver), 90
dobldobl_newton_power_series() (in module series), 139
dobldobl_newton_series() (in module series), 139
dobldobl_next_loop() (in module curves), 143
dobldobl_next_track() (in module curves), 143
dobldobl_pade_approximants() (in module series), 139
dobldobl_pade_coefficients() (in module curves), 143
dobldobl_pade_vector() (in module curves), 143
dobldobl_pole_radius() (in module curves), 143
dobldobl_pole_step() (in module curves), 143
dobldobl_poles() (in module curves), 143
dobld obl polysys_solve() (in module factor), 126
dobldobl_predict_correct() (in module curves), 143
dobldobl_random_coefficient_system() (in module solver), 90
dobldobl_random_system() (in module solver), 90
dobldobl_real_sweep() (in module sweepers), 107
dobldobl_retrieve() (in module tropisms), 114
dobldobl_scale_solutions() (in module curves), 143
dobldobl_scale_system() (in module solver), 90
dobldobl_scale_system() (in module curves), 143
dobldobl_series_coefficients() (in module tropisms), 114
dobldobl_series_step() (in module curves), 143
dobldobl_set_solution() (in module tropisms), 114
dobldobl_set_homotopy() (in module curves), 143
dobldobl_set_parameter_homotopy() (in module curves), 143
dobldobl_set_solution() (in module curves), 144
dobldobl_size() (in module tropisms), 115
dobldobl_solve() (in module solver), 90
dobldobl_start_diagonal_cascade() (in module diagonal), 128
dobldobl_step_size() (in module curves), 144
dobl_dobl_t_value() (in module curves), 144
dobl_dobl_track() (in module curves), 144
dobl_dobl_usolve() (in module solver), 90
double_double_cascade_step() (in module cascades), 124
double_double_crude_track() (in module trackers), 102
double_double_laurent_cascade_step() (in module cascades), 124
double_double_track() (in module trackers), 102
drop_coordinate_from_dobldobl_solutions() (in module sets), 118
drop_coordinate_from_quaddobl_solutions() (in module sets), 118
drop_coordinate_from_standard_solutions() (in module sets), 118
drop_variable_from_dobldobl_laurent_polynomials() (in module sets), 118
drop_variable_from_dobldobl_polynomials() (in module sets), 118
drop_variable_from_quaddobl_laurent_polynomials() (in module sets), 118
drop_variable_from_quaddobl_polynomials() (in module sets), 118
drop_variable_from_standard_laurent_polynomials() (in module sets), 118
drop_variable_from_standard_polynomials() (in module sets), 119

drop_coordinate_from_dobldobl_solutions() (in module sets), 118

drop_coordinate_from_quaddobl_solutions() (in module sets), 118

drop_coordinate_from_standard_solutions() (in module sets), 118

drop_variable_from_dobldobl_laurent_polynomials() (in module sets), 118

drop_variable_from_dobldobl_polynomials() (in module sets), 118

drop_variable_from_quaddobl_laurent_polynomials() (in module sets), 118

drop_variable_from_quaddobl_polynomials() (in module sets), 118

drop_variable_from_standard_laurent_polynomials() (in module sets), 118

drop_variable_from_standard_polynomials() (in module sets), 119

drop_coordinate_from_dobldobl_solutions() (in module sets), 118

drop_coordinate_from_quaddobl_solutions() (in module sets), 118

drop_coordinate_from_standard_solutions() (in module sets), 118

drop_variable_from_dobldobl_laurent_polynomials() (in module sets), 118

drop_variable_from_dobldobl_polynomials() (in module sets), 118

drop_variable_from_quaddobl_laurent_polynomials() (in module sets), 118

drop_variable_from_quaddobl_polynomials() (in module sets), 118

drop_variable_from_standard_laurent_polynomials() (in module sets), 118

drop_variable_from_standard_polynomials() (in module sets), 119

edges_in_facets() (in module polytopes), 137
embed() (in module sets), 119
endmultiplicity() (in module solutions), 98
evaluate() (in module solutions), 98
evaluate_polynomial() (in module solutions), 99
example4pade() (in module series), 140
examples
  module, 130

factor
  module, 126
  factor() (in module factor), 126
families
  module, 132
fbrfive4() (in module examples), 130
filter_real() (in module solutions), 99
filter_regular() (in module solutions), 99
filter_vanishing() (in module solutions), 99
filter_zero_coordinates() (in module solutions), 99
firsteqs() (in module families), 132
formdictlist() (in module solutions), 99

G
game4two() (in module examples), 130
generic_points, 54
generic_nash_system() (in module families), 132
get_corrector_residual_tolerance() (in module curves), 144
get_curvature_beta_factor() (in module curves), 144
get_degree_of.denominator() (in module curves), 144
get_degree_of.numerator() (in module curves), 144
get_gamma_constant() (in module curves), 144
get_homotopy_continuation_parameter() (in module curves), 144
get_maximum_corrector_steps() (in module curves), 144
get_maximum_step_size() (in module curves), 144
get_maximum_steps_on_path() (in module curves), 144
get_minimum_step_size() (in module curves), 145
get_pole_radius_beta_factor() (in module curves), 145
get_predictor_residual_alpha() (in module curves), 145
get_series_beta_factor() (in module curves), 145
get_zero_series_coefficient_tolerance() (in module curves), 145
gpu_double_double_track() (in module trackers), 103
gpu_double_track() (in module trackers), 103
gpu_quad_double_track() (in module trackers), 103

I
indeterminate_matrix() (in module families), 132
initial_form() (in module polytopes), 137
initial_support() (in module polytopes), 137
initialize_dobldobl_solution() (in module trackers), 103
initialize_dobldobl_tracker() (in module trackers), 103
initialize_multprec_solution() (in module trackers), 103
initialize_multprec_tracker() (in module trackers), 103
initialize_quaddobl_solution() (in module trackers), 103
initialize_quaddobl_tracker() (in module trackers), 104

Index 219
initialize_standard_solution() (in module trackers), 104
initialize_standard_tracker() (in module trackers), 104
initialize_varbprec_solution() (in module trackers), 104
initialize_varbprec_tracker() (in module trackers), 104
integer_mixed_cell() (in module polytopes), 137
integer_mixed_cells() (in module polytopes), 137
interface module, 150
is_binomial_system() (in module maps), 138
is_member() (in module sets), 119
is_real() (in module solutions), 99
is_signed() (in module sets), 119
is_slackvar() (in module sets), 119
is_square() (in module solver), 91
is_vanishing() (in module solutions), 99
ismember_filter() (in module sets), 119
K
katsura() (in module families), 132
katsura6() (in module examples), 130
katsura_variable() (in module families), 133
L
launchsolver() (in module dashboard), 149
laurent_cascade_filter() (in module cascades), 124
laurent_cascade_step() (in module cascades), 124
laurent_embed() (in module sets), 119
laurent_ismember_filter() (in module sets), 119
laurent_membertest() (in module sets), 120
laurent_top_cascade() (in module cascades), 124
laursys_solve() (in module factor), 126
linear_product_root_count() (in module solver), 91
linear_reduce() (in module solver), 91
littlewood_richardson_homotopies() (in module schubert), 134
load_dobldobl_laurent_system() (in module interface), 150
load_dobldobl_solutions() (in module interface), 150
load_dobldobl_system() (in module interface), 150
load_standard_laurent_system() (in module interface), 150
load_standard_solutions() (in module interface), 151
load_standard_system() (in module interface), 151
load_standard_tableau() (in module interface), 151
M
m_homogeneous_bezout_number() (in module solver), 91
m_homogeneous_start_system() (in module solver), 91
m_partition_bezout_number() (in module solver), 91
main() (in module schubert), 134
main() (in module server), 71
make_fractions() (in module series), 140
make_pieri_system() (in module schubert), 134
make_solution() (in module solutions), 99
management_test() (in module tropisms), 115
map_double() (in module solutions), 100
maps module, 138
max_steps_get() (in module tuning), 109
max_steps_set() (in module tuning), 109
maxnum_reruns_get() (in module tuning), 109
maxnum_reruns_set() (in module tuning), 109
membertest() (in module sets), 120
mixed_volume() (in module polytopes), 137
mixed_volume() (in module solver), 91
mixed_volume_by_demics() (in module solver), 91
module cascades, 123
curves, 142
dashboard, 149
diagonal, 128.examples, 130
factor, 126
families, 132
interface, 150
maps, 138
phcpy2c3, 154
polynomials, 101
polytopes, 136
schubert, 134
series, 139
server, 71
sets, 117
solutions, 98
solver, 89
sweepers, 106
trackers, 101
tropisms, 114
tuning, 108
monodromy_breakup() (in module factor), 126
monomial_map_solutions() (in module maps), 138
monomial_map_strings() (in module maps), 138
multiplicity() (solutions.Solution method), 98
multiprecision_track() (in module trackers), 104
multprec_usolve() (in module solver), 92

N
names_of_variables() (in module solver), 92
nash() (in module families), 133
nbodysys() (in module families), 133
newton_laurent_step() (in module solver), 92
newton_step() (in module solver), 92
next() (dashboard.Scroller method), 149
next_dobldobl_solution() (in module trackers), 104
next_multprec_solution() (in module trackers), 104
next_quaddobl_solution() (in module trackers), 104
next_standard_solution() (in module trackers), 104
next_varbprec_solution() (in module trackers), 104
noon() (in module families), 133
noon3() (in module examples), 130
number_of_symbols() (in module solver), 92
numerals() (in module solutions), 100
numerals() (solutions.Solution method), 98
numerical irreducible decomposition, 54

O
order_endgame_extrapolator_get() (in module tuning), 110
order_endgame_extrapolator_set() (in module tuning), 110
osculating_input() (in module schubert), 135

P
permute_dobldobl_system() (in module solver), 92
permute_quaddobl_system() (in module solver), 92
permute_standard_system() (in module solver), 92
phcpy2c3
module, 154
pieri_problem() (in module families), 133
pieri_root_count() (in module schubert), 135
planar_convex_hull() (in module polytopes), 137
planar_hull_checkout() (in module polytopes), 137
planes_to_string() (in module schubert), 135
plot() (dashboard.CoordinatePlot method), 149
plotcoordinate() (in module dashboard), 149
points_to_string() (in module schubert), 135
polegs() (in module families), 133
pols2str() (in module dashboard), 149
polynomials
module, 101
Polynomials (class in polynomials), 101
polysys_solve() (in module factor), 127
polytopes
module, 136
predictor_expfacendgame_get() (in module tuning), 110
predictor_expfacendgame_set() (in module tuning), 110
predictor_exfpaconpath_get() (in module tuning), 110
predictor_exfpaconpath_set() (in module tuning), 110
predictor_exptsendgame_get() (in module tuning), 110
predictor_exptsendgame_set() (in module tuning), 110
predictor_exptrsonpath_get() (in module tuning), 110
predictor_exptrsonpath_set() (in module tuning), 110
predictor_maxstepependgame_get() (in module tuning), 110
predictor_maxstepependgame_set() (in module tuning), 110
predictor_maxsteponpath_get() (in module tuning), 110
predictor_minstepependgame_get() (in module tuning), 111
predictor_minstepependgame_set() (in module tuning), 111

Index 221
predictor_minsteponpath_get() (in module tuning), 111
predictor_minsteponpath_set() (in module tuning), 111
predictor_redfacendgame_get() (in module tuning), 111
predictor_redfacendgame_set() (in module tuning), 111
predictor_redfaconpath_get() (in module tuning), 111
predictor_redfaconpath_set() (in module tuning), 111
predictor_typeendgame_get() (in module tuning), 112
predictor_typeendgame_set() (in module tuning), 112
predictor_typeonpath_get() (in module tuning), 112
predictor_typeonpath_set() (in module tuning), 112
previous() (dashboard.Scroller method), 149
prompt_for_dimensions() (in module schubert), 135
py2c_ade_manypaths_d() (in module phcpy2c3), 154
py2c_ade_manypaths_d_pars() (in module phcpy2c3), 154
py2c_ade_manypaths_dd() (in module phcpy2c3), 154
py2c_ade_manypaths_dd_pars() (in module phcpy2c3), 154
py2c_ade_manypaths_qd() (in module phcpy2c3), 154
py2c_ade_manypaths_qd_pars() (in module phcpy2c3), 154
py2c_ade_newton_d() (in module phcpy2c3), 155
py2c_ade_newton_dd() (in module phcpy2c3), 155
py2c_ade_newton_qd() (in module phcpy2c3), 155
py2c_ade_onepath_d() (in module phcpy2c3), 155
py2c_ade_onepath_dd() (in module phcpy2c3), 155
py2c_ade_onepath_qd() (in module phcpy2c3), 155
py2c_annote_autotune_continguation() (in module phcpy2c3), 155
py2c_celcon_append_lifted_point() (in module phcpy2c3), 155
py2c_celcon_clear_container() (in module phcpy2c3), 155
py2c_celcon_copy_into_dobldobl_systems() (in module phcpy2c3), 155
py2c_celcon_copy_into_dobldobl_systems_container() (in module phcpy2c3), 155
py2c_celcon_copy_into_quaddobl_systems_container() (in module phcpy2c3), 155
py2c_celcon_copy_into_standard_systems_container() (in module phcpy2c3), 155
py2c_celcon_dobldobl_polyhedral_homotopy() (in module phcpy2c3), 156
py2c_celcon_dobldobl_random_coefficient_system() (in module phcpy2c3), 156
py2c_celcon_dobldobl_system() (in module phcpy2c3), 156
py2c_celcon_is_stable() (in module phcpy2c3), 156
py2c_celcon_is_volume_of_cells() (in module phcpy2c3), 156
py2c_celcon_number_of_original_cells() (in module phcpy2c3), 156
py2c_celcon_number_of_cells() (in module phcpy2c3), 156
py2c_celcon_number_of_stable_cells() (in module phcpy2c3), 156
py2c_celcon_permute_dobldobl_system() (in module phcpy2c3), 156
py2c_celcon_permute_dobldobl_systems() (in module phcpy2c3), 156
py2c_celcon_permute_quaddobl_system() (in module phcpy2c3), 156
py2c_celcon_permute_quaddobl_systems() (in module phcpy2c3), 156
py2c_celcon_permute_standard_system() (in module phcpy2c3), 157
py2c_celcon_permute_standard_systems() (in module phcpy2c3), 157
py2c_celcon_quaddobl_polyhedral_homotopy() (in module phcpy2c3), 157
py2c_celcon_quaddobl_random_coefficient_system() (in module phcpy2c3), 157
py2c_celcon_quaddobl_random_coefficient_systems() (in module phcpy2c3), 157
py2c_celcon_retrieve_lifted_point() (in module phcpy2c3), 157
py2c_celcon_set_type_of_mixture() (in module phcpy2c3), 157
py2c_celcon_solve_dobldobl_start_system() (in module phcpy2c3), 157
py2c_celcon_solve_dobldobl_start_systems() (in module phcpy2c3), 157
py2c_celcon_solve_quaddobl_start_system() (in module phcpy2c3), 157
py2c_celcon_solve_quaddobl_start_systems() (in module phcpy2c3), 157
py2c_celcon_solve_standard_start_system() (in module phcpy2c3), 157
py2c_celcon_solve_standard_start_systems() (in module phcpy2c3), 157
py2c_celcon_standard_polyhedral_homotopy() (in module phcpy2c3), 158
py2c_celcon_standard_random_coefficient_system() (in module phcpy2c3), 158
py2c_celcon_standard_random_coefficient_systems() (in module phcpy2c3), 158
py2c_celcon_track_dobldobl_solution_path() (in module phcpy2c3), 158
Index
py2c_padcon_initialize_quaddobl_solution() (in module phcpy2c3), 178
py2c_padcon_initialize_standard_solution() (in module phcpy2c3), 178
py2c_padcon_quaddobl_closest_pole() (in module phcpy2c3), 178
py2c_padcon_quaddobl_denominator_coefficient() (in module phcpy2c3), 178
py2c_padcon_quaddobl_estimated_distance() (in module phcpy2c3), 179
py2c_padcon_quaddobl_hessian_step() (in module phcpy2c3), 179
py2c_padcon_quaddobl_initialize_homotopy() (in module phcpy2c3), 179
py2c_padcon_quaddobl_initialize_parameter_homotopy() (in module phcpy2c3), 179
py2c_padcon_quaddobl_numerator_coefficient() (in module phcpy2c3), 179
py2c_padcon_quaddobl_pole() (in module phcpy2c3), 179
py2c_padcon_quaddobl_pole_radius() (in module phcpy2c3), 179
py2c_padcon_quaddobl_pole_step() (in module phcpy2c3), 179
py2c_padcon_quaddobl_predict_correct() (in module phcpy2c3), 179
py2c_padcon_quaddobl_series_coefficient() (in module phcpy2c3), 179
py2c_padcon_quaddobl_series_step() (in module phcpy2c3), 179
py2c_padcon_quaddobl_step_size() (in module phcpy2c3), 179
py2c_padcon_quaddobl_t_value() (in module phcpy2c3), 179
py2c_padcon_quaddobl_track() (in module phcpy2c3), 179
py2c_padcon_set_default_parameters() (in module phcpy2c3), 180
py2c_padcon_set_homotopy_continuation_gamma() (in module phcpy2c3), 180
py2c_padcon_set_homotopy_continuation_parameter() (in module phcpy2c3), 180
py2c_padcon_standard_closest_pole() (in module phcpy2c3), 178
py2c_padcon_standard_denominator_coefficient() (in module phcpy2c3), 178
py2c_padcon_standard_estimated_distance() (in module phcpy2c3), 179
py2c_padcon_standard_hessian_step() (in module phcpy2c3), 179
py2c_padcon_standard_initialize_homotopy() (in module phcpy2c3), 179
py2c_padcon_standard_initialize_parameter_homotopy() (in module phcpy2c3), 180
py2c_padcon_standard_numerator_coefficient() (in module phcpy2c3), 179
py2c_padcon_standard_pole() (in module phcpy2c3), 179
py2c_padcon_standard_pole_radius() (in module phcpy2c3), 179
py2c_padcon_standard_pole_step() (in module phcpy2c3), 180
py2c_padcon_standard_predict_correct() (in module phcpy2c3), 180
py2c_padcon_standard_series_coefficient() (in module phcpy2c3), 180
py2c_padcon_standard_series_step() (in module phcpy2c3), 180
py2c_padcon_standard_step_size() (in module phcpy2c3), 180
py2c_padcon_standard_t_value() (in module phcpy2c3), 180
py2c_padcon_standard_track() (in module phcpy2c3), 180
py2c_PHCpack_version_string() (in module phcpy2c3), 154
py2c_product_clear_set_structure() (in module phcpy2c3), 181
py2c_product_is_set_structure_supporting() (in module phcpy2c3), 181
py2c_product_linear_product_root_count() (in module phcpy2c3), 181
py2c_product_m_homogeneous_Bezout_number() (in module phcpy2c3), 181
py2c_product_m_homogeneous_start_system() (in module phcpy2c3), 181
py2c_product_m_partition_Bezout_number() (in module phcpy2c3), 181
py2c_product_parse_set_structure() (in module phcpy2c3), 181
py2c_product_random_linear_product_system() (in module phcpy2c3), 182
py2c_product_set_structure_string() (in module phcpy2c3), 182
py2c_product_solve_linear_product_system() (in module phcpy2c3), 182
py2c_product_supporting_set_structure() (in module phcpy2c3), 182
py2c_product_write_set_structure() (in module phcpy2c3), 182
py2c_quaddobl_cascade_homotopy() (in module phcpy2c3), 183
py2c_quaddobl_collapse_diagonal() (in module phcpy2c3), 183
py2c_quaddobl_crude_tracker() (in module phcpy2c3), 183
py2c_quaddobl_deflate() (in module phcpy2c3),
py2c_quaddobl_diagonal_cascade_solutions() (in module phcpy2c3), 183
py2c_quaddobl_diagonal_homotopy() (in module phcpy2c3), 183
py2c_quaddobl_Laurent_cascade_homotopy() (in module phcpy2c3), 182
py2c_quaddobl_laursys_solve() (in module phcpy2c3), 183
py2c_quaddobl_multiplicity_structure() (in module phcpy2c3), 183
py2c_quaddobl_Newton_Laurent_step() (in module phcpy2c3), 182
py2c_quaddobl_Newton_power_series() (in module phcpy2c3), 182
py2c_quaddobl_Newton_series() (in module phcpy2c3), 182
py2c_quaddobl_Newton_step() (in module phcpy2c3), 182
py2c_quaddobl_Pade_approximant() (in module phcpy2c3), 182
py2c_quaddobl_polysys_solve() (in module phcpy2c3), 183
py2c_quaddobl_witset_of_hypersurface() (in module phcpy2c3), 184
py2c_quaddobl_witset_of_Laurent_hypersurface() (in module phcpy2c3), 183
py2c_read_dobldobl_start_Laurent_system() (in module phcpy2c3), 184
py2c_read_dobldobl_start_system() (in module phcpy2c3), 184
py2c_read_dobldobl_start_system_from_file() (in module phcpy2c3), 184
py2c_read_dobldobl_target_Laurent_system() (in module phcpy2c3), 184
py2c_read_dobldobl_target_system() (in module phcpy2c3), 184
py2c_read_dobldobl_target_system_from_file() (in module phcpy2c3), 184
py2c_read_quaddobl_start_Laurent_system() (in module phcpy2c3), 184
py2c_read_quaddobl_start_system() (in module phcpy2c3), 184
py2c_read_quaddobl_start_system_from_file() (in module phcpy2c3), 184
py2c_read_quaddobl_target_Laurent_system() (in module phcpy2c3), 184
py2c_read_quaddobl_target_system() (in module phcpy2c3), 184
py2c_read_quaddobl_target_system_from_file() (in module phcpy2c3), 184
py2c_read_standard_start_Laurent_system() (in module phcpy2c3), 185
py2c_read_standard_start_system() (in module phcpy2c3), 185
py2c_read_standard_start_system_from_file() (in module phcpy2c3), 185
py2c_read_standard_target_Laurent_system() (in module phcpy2c3), 185
py2c_read_standard_target_system() (in module phcpy2c3), 185
py2c_read_standard_target_system_from_file() (in module phcpy2c3), 185
py2c_scale_dobldobl_solutions() (in module phcpy2c3), 185
py2c_scale_dobldobl_system() (in module phcpy2c3), 185
py2c_scale_quaddobl_solutions() (in module phcpy2c3), 185
py2c_scale_quaddobl_system() (in module phcpy2c3), 185
py2c_scale_standard_solutions() (in module phcpy2c3), 186
py2c_scale_standard_system() (in module phcpy2c3), 186
py2c_scan_for_symbols() (in module phcpy2c3), 186
py2c_schubert_dobldobl_littlewood_richardson_homotopies() (in module phcpy2c3), 186
py2c_schubert_localization_poset() (in module phcpy2c3), 186
py2c_schubert_osculating_planes() (in module phcpy2c3), 186
py2c_schubert_pieri_count() (in module phcpy2c3), 186
py2c_schubert_pieri_homotopies() (in module phcpy2c3), 186
py2c_schubert_pieri_homotopies() (in module phcpy2c3), 186
py2c_schubert_pieri_homotopies() (in module phcpy2c3), 186
py2c_schubert_pieri_homotopies() (in module phcpy2c3), 186
py2c_schubert_pieri_homotopies() (in module phcpy2c3), 186
py2c_schubert_pieri_homotopies() (in module phcpy2c3), 186
py2c_schubert_pieri_homotopies() (in module phcpy2c3), 186
py2c_schubert_pieri_homotopies() (in module phcpy2c3), 186
py2c_schubert_pieri_homotopies() (in module phcpy2c3), 186
py2c_schubert_pieri_homotopies() (in module phcpy2c3), 186
py2c_schubert_pieri_homotopies() (in module phcpy2c3), 186
py2c_schubert_pieri_homotopies() (in module phcpy2c3), 186
py2c_schubert_pieri_homotopies() (in module phcpy2c3), 186
py2c_set_gamma_constant() (in module phcpy2c3), 187
py2c_set_seed() (in module phcpy2c3), 188
py2c_set_value_of_continuation_parameter() (in module phcpy2c3), 188
py2c_show_continuation_parameters() (in module phcpy2c3), 188
py2c_solcon_append_dobldobl_solution_string() (in module phcpy2c3), 188
py2c_solcon_append_multprec_solution_string() (in module phcpy2c3), 188
py2c_solcon_append_quaddobl_solution_string() (in module phcpy2c3), 188
py2c_solcon_append_quaddobl_solution_string() (in module phcpy2c3), 188
py2c_solcon_append_quaddobl_solution_string() (in module phcpy2c3), 188
py2c_solcon_append_standard_solution_string
(py2c_solcon_quaddobl_drop_coordinate_by_name)
(py2c_solcon_quaddobl_drop_coordinate_by_index)
(py2c_solcon_quaddobl_one_affinization)
(py2c_solcon_quaddobl_one_homogenization)
(py2c_solcon_read_dobldobl_solutions)
(py2c_solcon_read_dobldobl_solutions_from_file)
(py2c_solcon_read_multprec_solutions)
(py2c_solcon_read_quaddobl_solutions)
(py2c_solcon_read_quaddobl_solutions_from_file)
(py2c_solcon_read_standard_solutions)
(py2c_solcon_read_standard_solutions_from_file)
(py2c_solcon_retrieve_next_dobldobl_initialize)
(py2c_solcon_retrieve_next_dobldobl_solution_string)
(py2c_solcon_retrieve_next_dobldobl_to_next)
(py2c_solcon_retrieve_next_dobldobl_to_next_standard)
(py2c_solcon_retrieve_next_multprec_initialize)
(py2c_solcon_retrieve_next_multprec_solution_string)
(py2c_solcon_retrieve_next_quaddobl_initialize)
(py2c_solcon_retrieve_next_quaddobl_solution_string)
(py2c_solcon_retrieve_next_quaddobl_to_next)
(py2c_solcon_retrieve_next_quaddobl_to_next_standard)
(py2c_solcon_retrieve_next_standard_initialize)
(py2c_solcon_retrieve_next_standard_solution_string)
(py2c_solcon_retrieve_next_standard_to_next)
(py2c_solcon_standard_drop_coordinate_by_index)
(py2c_solcon_standard_drop_coordinate_by_name)
(py2c_solcon_standard_one_affinization)
(py2c_solcon_standard_one_homogenization)
(py2c_solcon_write_current_dobldobl_solution_string)
(py2c_solcon_write_current_dobldobl_solutions)
(py2c_solcon_write_current_dobldobl_to_next)
(py2c_solcon_write_current_dobldobl_to_next_standard)
(py2c_solcon_write_current_multprec_solution_string)
(py2c_solcon_write_current_multprec_solutions)
(py2c_solcon_write_current_quaddobl_solution_string)
(py2c_solcon_write_current_quaddobl_solutions)
(py2c_solcon_write_current_standard_solution_string)
(py2c_solcon_write_current_standard_solutions)
(py2c_solcon_write_standard_dobldobl_solution_string)
(py2c_solcon_write_standard_dobldobl_solutions)
(py2c_solcon_write_standard_multprec_solution_string)
(py2c_solcon_write_standard_multprec_solutions)
(py2c_solcon_write_standard_quaddobl_solution_string)
(py2c_solcon_write_standard_quaddobl_solutions)
(py2c_solcon_write_standard_standard_solution_string)
(py2c_solcon_write_standard_standard_solutions)
Index 231
Index
quaddobl_diagonal_solver() (in module diagonal), 129
quaddobl_dimension() (in module tropisms), 115
quaddobl_embed() (in module sets), 120
quaddobl_estimated_distance() (in module curves), 145
quaddobl_get_solution() (in module curves), 145
quaddobl_hessian_step() (in module curves), 145
quaddobl_initialize() (in module tropisms), 115
quaddobl_ismember() (in module sets), 120
quaddobl_ismember_filter() (in module sets), 120
quaddobl_laurent_embed() (in module sets), 121
quaddobl_laurent_ismember() (in module sets), 121
quaddobl_laurent_ismember_filter() (in module sets), 121
quaddobl_laurent_membertest() (in module sets), 121
quaddobl_laursys_solve() (in module factor), 127
quaddobl_linear_reduction() (in module solver), 93
quaddobl_littlewood_richardson_homotopies() (in module schubert), 135
quaddobl_membertest() (in module sets), 121
quaddobl_monodromy_breakup() (in module factor), 127
quaddobl_multiplicity() (in module solver), 93
quaddobl_newton_power_series() (in module series), 140
quaddobl_newton_series() (in module series), 140
quaddobl_next_loop() (in module curves), 145
quaddobl_next_track() (in module curves), 145
quaddobl_pade_approximants() (in module series), 141
quaddobl_pade_coefficients() (in module series), 145
quaddobl_pade_vector() (in module curves), 146
quaddobl_pole_radius() (in module curves), 146
quaddobl_pole_step() (in module curves), 146
quaddobl_poles() (in module curves), 146
quaddobl_polysys_solve() (in module factor), 127
quaddobl_predict_correct() (in module curves), 146
quaddobl_random_coefficient_system() (in module solver), 93
quaddobl_random_system() (in module solver), 93
quaddobl_real_sweep() (in module sweepers), 107
quaddobl_retrieve() (in module tropisms), 115
quaddobl_scale_solutions() (in module solver), 93
quaddobl_scale_system() (in module solver), 93
quaddobl_series_coefficients() (in module curves), 146
quaddobl_series_step() (in module curves), 146
quaddobl_set_homotopy() (in module curves), 146
quaddobl_set_parameter_homotopy() (in module curves), 146
quaddobl_set_solution() (in module curves), 146
quaddobl_size() (in module tropisms), 115
quaddobl_solve() (in module solver), 94
quaddobl_start_diagonal_cascade() (in module diagonal), 129
quaddobl_step_size() (in module curves), 146
quaddobl_t_value() (in module curves), 146
quaddobl_track() (in module curves), 146
quaddobl_usolve() (in module solver), 94
real_osculating_planes() (in module schubert), 135
real_random_trinomials() (in module solver), 94
real_sweep_test() (in module sweepers), 107
recpol() (in module families), 133
recv_strings() (in module server), 71
replace_symbol() (in module series), 141
resolve_schubert_conditions() (in module schubert), 135
retrieve_dobldobl_tropism() (in module tropisms), 115
retrieve_quaddobl_tropism() (in module tropisms), 115
retrieve_standard_tropism() (in module tropisms), 115
rps10() (in module examples), 130
run() (server.ServerHandler method), 71
run_cascade() (in module cascades), 125
run_pieri_homotopies() (in module schubert), 135

schubert
module, 134
Scroller (class in dashboard), 149
scrollsols() (in module dashboard), 149
send_strings() (in module server), 73
series
module, 139
server
module, 71
server_connect() (in module server), 73
ServerHandler (class in server), 71
set_homotopy_continuation() (in module curves), 147
set_homotopy_continuation_gamma() (in module curves), 147
set_homotopy_continuation_parameter() (in module curves), 147
set_path_parameter_value() (in module tuning), 113
sets
module, 117
sevenbar() (in module examples), 131
show() (dashboard.Scroller method), 149
Solution (class in solutions), 98
solutions
module, 98
solve() (dashboard.SolveButton method), 149
solve() (in module factor), 127
solve() (in module solver), 95
solve() (polynomials.Polynomials method), 101
solve_binomials() (in module examples), 131
solve_binomials() (in module maps), 138
solve_checkin() (in module solver), 95
solve_cyclic7() (in module examples), 131
solve_fbrfive4() (in module examples), 131
solve_game4two() (in module examples), 131
solve_katsura6() (in module examples), 131
solve_noon3() (in module examples), 131
solve_rps10() (in module examples), 131
solve_sevenbar() (in module examples), 131
solve_system() (in module server), 73
solve_tangents() (in module examples), 131
SolveButton (class in dashboard), 149
solver
module, 89
split_filter() (in module cascades), 125
standard_clear() (in module tropisms), 115
standard_closest_pole() (in module curves), 147
standard_complex_sweep() (in module sweepers), 107
standard_condition_report() (in module solver), 95
standard_decomposition() (in module factor), 127
standard_deflate() (in module solver), 95
standard_diagonal_cascade_solutions() (in module diagonal), 129
standard_diagonal_homotopy() (in module diagonal), 129
standard_diagonal_solver() (in module diagonal), 129
standard_dimension() (in module tropisms), 115
standard_double_cascade_step() (in module cascades), 125
standard_double_crude_track() (in module trackers), 105
standard_double_laurent_cascade_step() (in module cascades), 125
standard_double_track() (in module trackers), 105
standard_embed() (in module sets), 121
standard_estimated_distance() (in module curves), 147
standard_get_solution() (in module curves), 147
standard_hessian_step() (in module curves), 147
standard_initialize() (in module tropisms), 115
standard_ismember() (in module sets), 121
standard_ismember_filter() (in module sets), 122
standard_laurent_embed() (in module sets), 122
standard_laurent_ismember() (in module sets), 122

236
standard_laurent_ismember_filter()  (in module sets), 122
standard_laurent_membertest()  (in module sets), 122
standard_laursys_solve()  (in module factor), 127
standard_linear_reduction()  (in module solver), 95
standard_littlewood_richardson_homotopies()  (in module schubert), 136
standard_membertest()  (in module sets), 123
standard_monodromy_breakup()  (in module factor), 127
standard_multiplicity()  (in module solver), 95
standard_newton_power_series()  (in module series), 141
standard_newton_series()  (in module series), 141
standard_next_loop()  (in module curves), 147
standard_next_track()  (in module curves), 147
standard_nonlinear_reduction()  (in module solver), 96
standard_pade_approximants()  (in module series), 142
standard_pade_coefficients()  (in module curves), 147
standard_pade_vector()  (in module curves), 147
standard_pole_radius()  (in module curves), 147
standard_pole_step()  (in module curves), 147
standard_poles()  (in module curves), 127
standard_polysys_solve()  (in module factor), 127
standard_predict_correct()  (in module curves), 148
standard_random_coefficient_system()  (in module solver), 96
standard_random_system()  (in module solver), 96
standard_real_sweep()  (in module sweepers), 107
standard_retrieve()  (in module tropisms), 116
standard_scale_solutions()  (in module solver), 96
standard_scale_system()  (in module solver), 96
standard_series_coefficients()  (in module curves), 148
standard_series_step()  (in module curves), 148
standard_set_homotopy()  (in module curves), 148
standard_set_parameter_homotopy()  (in module curves), 148
standard_set_solution()  (in module curves), 148
standard_size()  (in module tropisms), 116

standard_solve()  (in module solver), 96
standard_start_diagonal_cascade()  (in module diagonal), 129
standard_step_size()  (in module curves), 148
standard_t_value()  (in module curves), 148
standard_track()  (in module curves), 148
standard_usolve()  (in module solver), 96
start_server()  (in module server), 73
store_dobldobl_laurent_system()  (in module interface), 151
store_dobldobl_laurent_witness_set()  (in module interface), 151
store_dobldobl_solutions()  (in module interface), 152
store_dobldobl_system()  (in module interface), 152
store_dobldobl_tropism()  (in module tropisms), 116
store_dobldobl_witness_set()  (in module interface), 152
store_laurent_system()  (in module maps), 138
store_multprec_laurent_system()  (in module interface), 152
store_multprec_solutions()  (in module interface), 152
store_multprec_system()  (in module interface), 152
store_quaddobl_laurent_system()  (in module interface), 152
store_quaddobl_laurent_witness_set()  (in module interface), 152
store_quaddobl_solutions()  (in module interface), 152
store_quaddobl_system()  (in module interface), 152
store_quaddobl_tropism()  (in module tropisms), 116
store_quaddobl_witness_set()  (in module interface), 153
store_standard_laurent_system()  (in module interface), 153
store_standard_laurent_witness_set()  (in module interface), 153
store_standard_solutions()  (in module interface), 153
store_standard_system()  (in module interface), 153
store_standard_tableau()  (in module interface), 153
store_standard_tropism()  (in module tropisms), 116
store_standard_witness_set()  (in module interface), 153

Index 237
str2complex() (in module solutions), 100
str2pols() (in module dashboard), 150
string_complex() (in module solutions), 100
string_coordinates() (in module solutions), 100
strsol2dict() (in module solutions), 100
strvar() (in module families), 133
substitute_symbol() (in module series), 142
support() (in module polytopes), 137
support_sets() (in module polytopes), 138
sweepers
  module, 106
symbolic_pade_approximant() (in module curves), 148
symbolic_pade_vector() (in module curves), 148
sysdi() (in module examples), 131

tangents() (in module examples), 131
test() (in module cascades), 125
test() (in module diagonal), 129
test() (in module examples), 132
test() (in module factor), 128
test() (in module families), 133
test() (in module interface), 154
test() (in module maps), 138
test() (in module polynomials), 101
test() (in module series), 142
test() (in module sets), 123
test() (in module solver), 96
test() (in module trackers), 105
test() (in module tropisms), 116
test() (in module tuning), 113
test_ade_double_double_track() (in module trackers), 105
test_ade_double_track() (in module trackers), 105
test_ade_quad_double_track() (in module trackers), 105
test_cascade() (in module cascades), 125
test_class() (in module solutions), 100
test_convex_hull() (in module polytopes), 138
test_crude_tracker() (in module trackers), 105
test_decompose() (in module factor), 128
test_deflate() (in module solver), 97
test_diaghom() (in module diagonal), 129
test_dobldobl_polyhedral_homotopy() (in module solver), 97
test_dobldobl_store_load() (in module tropisms), 116
test_factor() (in module factor), 128
test_functions() (in module solutions), 100
test_hyperbola() (in module curves), 148
test_integer_mixed_volume() (in module polytopes), 138
tolerance_infsolonpath_get() (in module tuning), 113
tolerance_infsolonpath_set() (in module tuning), 113
tolerance_rcondendgame_get() (in module tuning), 113
tolerance_rcondendgame_set() (in module tuning), 113
tolerance_rcondonpath_get() (in module tuning), 113
tolerance_rcondonpath_set() (in module tuning), 113
top_cascade() (in module cascades), 125
top_diagonal_dimension() (in module diagonal), 129
total_degree() (in module solver), 97
total_degree_start_system() (in module solver), 97
track() (in module trackers), 106
track_simultaneously_get() (in module tuning), 114
track_simultaneously_set() (in module tuning), 114
trackers module, 101
tropisms module, 114
tune_homotopy_continuation_parameters() (in module curves), 149
tune_path_parameters() (in module tuning), 114
tune_track_parameters() (in module tuning), 114
tuning module, 108

U
usolve() (in module solver), 97

V
variables() (in module solutions), 100
variables() (polynomials.Polynomials method), 101
variables() (solutions.Solution method), 98
verify() (in module schubert), 136
vertices_in_facets() (in module polytopes), 138
viviani() (in module series), 142
viviani2() (in module series), 142

W
windowsize() (in module dashboard), 150
witness_set_of_hypersurface() (in module sets), 123
witness_set_of_laurent_hypersurface() (in module sets), 123
write_decomposition() (in module factor), 128
write_homotopy_continuation_parameters() (in module curves), 149
write_monomial_map() (in module maps), 138
write_monomial_maps() (in module maps), 139
write_path_parameters() (in module tuning), 114

Index 239