PHClab: A MATLAB/Octave Interface to PHCpack[*]

Yun Guan[†] Jan Verschelde[‡]

6 March 2007

Abstract

PHCpack is a software package for Polynomial Homotopy Continuation, to numerically solve systems of polynomial equations. The executable program "phc" produced by PHCpack has several options (the most popular one "-b" offers a blackbox solver) and is menu driven. PHClab is a collection of scripts which call phc from within a MATLAB or Octave session. It provides an interface to the blackbox solver for finding isolated solutions. We executed the PHClab functions on our cluster computer using the MPI ToolBox (MPITB) for Octave to solve a list of polynomial systems. PHClab also interfaces to the numerical irreducible decomposition, giving access to the tools to represent, factor, and intersect positive dimensional solution sets.

1 Introduction

Polynomial systems arise in various fields of science and engineering, e.g.: the design of a robot arm [13] so its hands passes through a prescribed sequence of points in space requires the solution of a polynomial system. Homotopy continuation methods are efficient numerical algorithms to approximate all isolated solutions of a polynomial system [10]. Recently homotopies have been developed to describe positive dimensional solution sets [20].

This paper documents an interface PHClab to use the functionality provided by PHCpack [21] from within a MATLAB or Octave session. The main executable program provided by PHCpack is `phc`, available for downloading on a wide variety of computers and operating systems. The program phc requires no compilation. Its most popular mode of

[*]This material is based upon work supported by the National Science Foundation under Grant No. 0105739 and Grant No. 0134611.

[†]Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago, 851 South Morgan (M/C 249), Chicago, IL 60607-7045, USA. *Email:* guan@math.uic.edu.

[‡]Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago, 851 South Morgan (M/C 249), Chicago, IL 60607-7045, USA. *Email:* jan@math.uic.edu or jan.verschelde@na-net.ornl.gov. *URL:* http://www.math.uic.edu/~jan.

operation is via the blackbox option, i.e.: as `phc -b input output`. Recently the program has been updated with tools to compute a numerical irreducible decomposion [18].

The main motivation for PHClab is to make it easier to use `phc` by automatic conversions of the formats for polynomial systems (on input) and solutions (on output). Manual or adhoc conversions can be tedious and lead to errors. As PHCpack has no scripting language on its own, the second advantage of PHClab is help the user to systematically use the full capabilities of PHCpack. As MATLAB (and its freely available counterpart Octave) is a very popular scientific software system, PHClab will be a useful addition to PHCpack.

Another feature of PHClab is the possibility of developing parallel code at a high level, when using the MPI ToolBox (MPITB [2]) for Octave.

Acknowledgments. This paper was written while the second author was a long term visitor at the IMA in the Fall of 2006, participating to the annual thematic program on Applications of Algebraic Geometry. The first author is grateful for financial support to present this work at the IMA software workshop, held at the IMA, 23-27 October, 2006. The first author also thanks Javier Fernández Baldomero, the author of MPITB, and Michael Creel for helpful communication. We thank Bernd Sturmfels for forwarding origami equations [1] to the IMA workshop "Software for Algebraic Geometry" as a challenge problem.

2 The Design of PHClab

The first interface from a C program to `phc` was written by Nobuki Takayama and is still available via the PHCpack download web site. Via this interface, `phc` became part of OpenXM [11] (see also [12]). We used the same idea to build `PHCmaple` [7, 8], defining a Maple interface to PHCpack.

All that is needed to make the interface work is the executable form of `phc`. PHClab is a collection of scripts written in the language of MATLAB and Octave. These scripts call `phc` with the appropriate options and menu choices.

3 Downloading and Installing

PHClab was tested on Matlab 6.5 and Octave 2.1.64 on computers running Windows and Linux. On an Apple laptop running Mac OS X version 10.3.7, we executed PHClab in Octave 2.1.57.

The most recent version of PHCpack and PHClab can be retrieved from

`http://www.math.uic.edu/~jan/download.html` which we from now on call the download web site. To install and use PHClab, execute the following steps:

1. From the download web site, either download the source code for `phc` (a makefile is provided with the code), or select an executable version of `phc`. Currently, `phc` is available in executable form on Windows, workstations from IBM (running AIX 5.3) and SUN (running SunOS 5.8), and PCs running Linux and Mac OS X 10.3. Except for Windows (which comes just as a plain `phc.exe`), one has to run `gunzip` followed by `tar xpf` on the downloaded file.

2. The PHClab distribution is available as `PHClab.tar.gz` from the download web site. To install PHClab in the directory /tmp, save `PHClab.tar.gz` first in `/tmp`, and then execute the following sequence of commands: `cd /tmp`; `mkdir PHClab`; `mv /tmp/PHClab.tar.gz PHClab`; `cd /tmp/PHClab`; `gunzip PHClab.tar.gz`; and finally: `tar xpf PHClab.tar`.

3. Either launch MATLAB or Octave in the directory PHClab, or add the name of the directory which contains PHClab to the path of MATLAB or Octave.

The first command of PHClab which one must execute is **set_phcpath**. This command takes one argument: the full path name of the file name which contains the executable program `phc`. For example, if `phc` was saved in `/tmp`, then a session with PHClab must start with **set_phcpath**(`'/tmp/phc'`).

4 Solving Polynomial Systems

In this section we define the basic commands to solve polynomial systems using PHClab. We first define the input/output formats, introducing the function **make_system** to convert a matrix format for a polynomial system into a symbolic input format to `phc`. The blackbox solver of PHCpack is called by the command **solve_system**. Besides the solution vectors, the solver returns extra diagnostical information about the quality of each solution.

Path tracking typically starts from a generic system (without any singular solutions) to a more specific system. We use the system we first solved by the blackbox solver as start system to solve a system with specific coefficients, using the function **track**. Because of our specific choice of the coefficients, we generated a polynomial system with a double solution, i.e.: of multiplicity two. Via the function **refine_sols** and **deflation**, we respectively refine a solution and deflate its multiplicity into a regular problem.

The last PHClab function we introduce in this section is **mixed_volume**, to compute the mixed volume for a polynomial system and (optionally) create and solve a random coefficient start system. The mixed volume equals the number of roots without zero components of a polynomial system with sufficiently generic coefficients. The function **mixed_volume** calls the translated form of the code MixedVol [3].

4.1 I/O Formats and the Blackbox Solver

The input to the solver is a system of multivariate equations with complex floating-point coefficients. For example, consider the system $g(\mathbf{x}) = \mathbf{0}$:

$$g(x_1, x_2) = \left\{ \begin{array}{r} 1.3x_1^2 + 4.7x_2^2 - 3.1 + 2.3i = 0 \\ 2.1x_2^2 - 1.9x_1 = 0 \end{array} \right. , \quad \text{with } i = \sqrt{-1}. \tag{1}$$

This system is encoded as a matrix, with in its rows the terms of each polynomial. A zero row in the matrix marks the end of a polynomial in the system. A nonzero row in the matrix represents a term as the coefficient followed by the exponents for each variable. For example $4.7x_2^2$ is represented by the row `4.7 0 2`. If n is the number of variables and m the total number of terms, then the matrix encoding the system has $m+n$ rows and $n+1$ columns.

To solve the system $g(\mathbf{x}) = \mathbf{0}$ using PHClab, we may execute the following sequence of instructions:

```
% tableau input for a system :
t = [1.3 2 0; 4.7 0 2; -3.1 + 2.3*i 0 0; 0 0 0;
     2.1 0 2; -1.9 1 0; 0 0 0];
make_system(t)               % shows symbolic format of the system
s = solve_system(t);         % call the blackbox solver
ns = size(s,2)               % check the number of solutions
s3 = s(3)                    % look at the 3rd solution
```

Then we see the following output on screen:

```
ans =

    ' + 1.3*x1**2 + 4.7*x2**2 + (-3.1+2.3*i)'
    ' + 2.1*x2**2 -1.9*x1'



ns =

     4



s3 =


            time: 1
    multiplicity: 1
             err: 4.0340e-16
```

```
rco: 0.1243
res: 2.7760e-16
 x1: -3.9180 + 0.3876i
 x2: 0.0930 + 1.8851i
```

We see the coordinates of the solution are in the last fields (displayed by default in short format, we may see more in format long) and extra diagnostics in the first five fields, briefly explained below.

time is the end value of the continuation parameter. If this value is not equal to one, then it means that the path tracker did not manage to reach the end of the path. This may happens with paths diverging to infinity or with highly singular solutions.

multiplicity is the multiplicity of the solution. A solution is regular when the multiplicity is one. When the approximation for a solution is not yet accurate enough, then the multiplicity might still be reported as one, although the value for `rco` might be close to the threshold.

err is the magnitude of the last update Newton's method made to the solution. At singular solutions, the polynomial functions exhibit a typical "flat" behavior. Although the residual may then be already very small, the value for this `err` can be still large.

rco is an estimate for the inverse of a condition number of the Jacobian matrix evaluated at the approximate solution. A solution is deemed singular when this number drops below the threshold value of 10^{-8}. Multiple solutions are singular. The condition number C of the Jacobian matrix measures the forward error, i.e.: if the coefficients are given with D digits precision, then the error on the approximate solution can be as large as $C \times 10^{-D}$.

res is the residual, or the magnitude of the polynomial system evaluated at the approximate solution. This residual measures the backward error: how much one should change the coefficients of the given system to have the computed approximation as the exact solution.

The values of the coordinates of the solutions are by default displayed in MATLAB's (or Octave's) `format short`. By `format long e` we can see the full length in scientific format. For the solution above, the values of `err`, `rco`, and `res` indicate an excellent quality of the computed solution.

4.2 Path Tracking from a Generic to a Specific System

The four solutions of the system we solved are all very well conditioned, so we may use them as start solutions to solve a system with the same coefficient structure, but with more specific coefficients:

$$f(x_1,x_2) = \left\{ \begin{array}{r} x_1^2 + 4x_2^2 - 4 = 0 \\ -2x_2^2 + x_1 - 2 = 0 \end{array} \right. , \quad \text{with } i = \sqrt{-1}. \tag{2}$$

Geometrically, the polynomials in the system $f(\mathbf{x}) = \mathbf{0}$ respectively define an ellipse and a parabola, positioned in such a way that their real intersection point is a double solution.

In the sequence of instructions below we use the function **track**, using the new system `double` (the system $f(\mathbf{x}) = \mathbf{0}$) as target system and the system `t` we solved as start system (we called it $g(\mathbf{x}) = \mathbf{0}$). Note that before calling **track**, we must set the value of time in every solution to zero, so `s` contains proper start solutions.

```
double = [1.0 2 0; 4.0 0 2; -4.0 0 0; 0 0 0;
          -2.0 0 2; +1.0 1 0; -2.0 0 0; 0 0 0];
make_system(double)             % shows symbolic format of the system
s(1).time = 0; s(2).time = 0;   % initialize time for every start
s(3).time = 0; s(4).time = 0;   % solution to zero
sols = track(double,t,s);       % call the blackbox solver
ns = size(sols,2)               % check the number of solutions
s2 = sols(2)                    % look at the 2nd solution
```

The choice of the second solution was done on purpose because this solution needs extra processing. In general however, we have no control over the order in which the solutions are computed, i.e.: while every run should give the same four solutions back, the order of solutions could be permuted.

The output we see on screen of the sequence above is

```
ans =

    'x1**2 + 4*x2**2 -4'
    ' -2*x2**2 + x1 -2'



ns =

     4



s2 =
          time: 1
  multiplicity: 1
           err: 4.373000000000000e-07
           rco: 3.147000000000000e-07
           res: 7.235000000000000e-13
            x1: 2.000000000000000e+00 - 5.048709793414480e-29i
            x2: -2.493339146012010e-07 - 1.879166705634450e-07i
```

Recall that we constructed the equations in our second system $f(\mathbf{x}) = \mathbf{0}$ so that there is a double solution at $(2, 0)$. However, since we are not yet close enough to the actual double solution $(2, 0)$, the magnitude of the condition number is about 10^7, below the threshold of 10^8, so `phc` does not recognize the solution as a double root. We will next see how to get closer to the actual solution.

4.3 Refining and Reconditioning Singular Solutions

To refine the solution we save in `s2`, we execute 10 addition Newton steps, applying **refine_sols** to the second solution `s2`:

```
r2 = refine_sols(double,s2,1.0e-16,1.0e-08,1.0e-16,10)
```

We allow 10 iterations (last parameter of **refine_sols**) of Newton's method, requiring that either the magnitude of the correction vector (`err`) or the residual (`res`) is less or equal than 10^{-16}, as specified respectively by the third and fifth parameter of **refine_sols**.

Below, on the output we see the estimate for the inverse condition number has decreased, along with the value for `x2`:

```
r2 =


          time: 1
  multiplicity: 1
           err: 3.300000000000000e-09
           rco: 3.885000000000000e-09
           res: 6.427999999999999e-17
            x1: 2.000000000000000e+00 + 4.309100000000000e-41i
            x2: -2.999062183346541e-09 - 3.017695139191104e-10i
```

Now that the estimate for the inverse condition number has dropped from 10^{-7} to 10^{-9}, below the threshold of 10^{-8}, we expect this solution to be singular. To deflate the multiplicity [9] and recondition the solution, we execute

```
def_sols = deflation(double,sols);
def_sols{4,1}
```

and then we see on screen

```
ans =

            time: 1
    multiplicity: 2
             err: 2.186000000000000e-07
             rco: 1.242000000000000e-01
             res: 1.003000000000000e-13
              x1: 2.000000000000010e+00 + 1.929286255918420e-14i
              x2: -1.742621478521780e-14 + 8.266179457715231e-15i
          lm_1_1: 3.077939801899640e-01 + 6.678691166401400e-01i
          lm_1_2: -6.737524546080300e-01 - 2.946929268111410e-01i
```

Notice the value `rco` which has increased dramatically from `3.885e-09` to `1.242e-01`, as a clear indication that the solution returned by deflation is well conditioned. Yet the multiplicity is two as a solution of the original system. The deflation procedure has constructed an augmented system for which the double solution of the original system is a regular root. The values for `lm_1_1` and `lm_1_2` are the values of the multipliers $\lambda_{1,1}$ and $\lambda_{1,2}$ used in the first deflation of the system. The number of multipliers used equals the one plus the numerical rank of the given approximate solution evaluated at the Jacobian matrix of the original system. The augmented system is returned in `def_sols{3,1}`.

4.4 Mixed Volumes and Random Coefficient Systems

To solve the system (2) we used the output of the blackbox solver on a more general system. The blackbox solver uses polyhedral homotopies [5] to solve a system with the same sparse structure but with random coefficients. Such random coefficient system has exactly as many isolated solutions as its mixed volume [10]. The function **mixed_volume** in PHClab gives access to the code MixedVol [3] as it is available as translated form in PHCpack.

If we continue our session with in `double` the tableau input for the system (2), then we can compute its mixed volume and solve a random coefficients start system via the following sequence of commands:

```
[v,g,s] = mixed_volume(double,1);   % call the blackbox solver
v                                   % check the mixed volume
ns = size(s,2)                      % check the number of solutions
g                                   % see the random coefficient system
```

The output to these command is

```
v = 4
```

```
ns = 4
g =

{
  [1,1] = +( 9.51900029533701E-01 + 3.06408769087537E-01*i)*x1^2
          +( 9.94012861166580E-01 + 1.09263131180786E-01*i)
  [2,1] = +( 6.10442645118414E-01 - 7.92060462982993E-01*i)*x2^2
          +(-5.76175858933274E-01 - 8.17325748757804E-01*i)
}
```

5 Solving Many Systems

Using PHCpack from within a MATLAB or Octave session provides novel opportunities to solve polynomial systems. In this section we show how the scripting environments can help to control the quality of the developed software. The high level parallel programming capabilities of MPITB will speed up this process in a convenient manner.

5.1 Automatic Testing and Benchmarking

The scripting language of MATLAB and Octave lends itself very directly to automatically solving many polynomial systems, as one would do for benchmarking purposes.

We introduce another PHClab function: **read_system** which reads a polynomial system from file. The value returned by this function can be passed to the blackbox solver. The system on file must have the following format. On the first line we have two numbers: the number of equations and variables. Thereafter follow the polynomials, each one is terminated by a semicolon. For example, the system $g(\mathbf{x}) = \mathbf{0}$ is represented as

```
2 2
 1.3*x1**2 + 4.7*x2**2 + (-3.1+2.3*i);
 2.1*x2**2 -1.9*x1;
```

Note that `i` (and `I`) may not be used to denote variables, as they both represent the imaginary unit $\sqrt{-1}$. Because `e` and `E` are used to denote floating-point numbers, `e` and `E` may not by used as the start of names of variables.

If `/tmp/Demo` contains the polynomial systems in the files with names `ku10`, `cyclic5`, `/tmp/Demo/fbrfive4`, `/tmp/Demo/game4two`, (taken from the demonstration database[1] at [21]), then the script with contents

[1]available at `http://www.math.uic.edu/~jan/demo.html`

```
f = {'/tmp/Demo/ku10'
     '/tmp/Demo/cyclic5'
     '/tmp/Demo/fbrfive4'
     '/tmp/Demo/game4two'};
for k= 1:size(f,1)
  p = read_system(f{k});
  t0 = clock;
  s = solve_system(p);
  et = etime(clock(),t0);
  n = size(s,2);
  fprintf('Found %d solutions for %s in %f seconds.\n',n,f{k},et);
end;
```

will produce the following statistics:

```
Found 2 solutions for /tmp/Demo/ku10 in 1.819892 seconds.
Found 70 solutions for /tmp/Demo/cyclic5 in 11.094403 seconds.
Found 36 solutions for /tmp/Demo/fbrfive4 in 18.750158 seconds.
Found 9 solutions for /tmp/Demo/game4two in 1.630962 seconds.
```

5.2 Parallel Scripting with MPITB

MPITB for Octave [2] extends Octave environment by using DLD functions. It allows Octave users in a computer cluster to build message-passing based parallel applications, by the means of installing the required packages and adding MPI calls to Octave scripts. To use MPITB for Octave, dynamically linked LAM/MPI libraries are required. All nodes in the cluster need to be able to access the custom-compiled Octave that supports DLD functions.

Our choice of MPITB for Octave was motivated primarily by its functionality and availability through open source. In our testing environment, the latest MPITB for Octave was compiled against LAM/MPI 7.1.2 and Octave 2.1.64. To illustrate conducting parallel computation with the combination of MPITB, PHClab and Octave, we used origami equations [1]. The main script is small enough to be included here:

```
function origami
%
% mpirun -c <nprocs> octave-2.1.64 -q --funcall origami
%
% The manager distributes the systems to the worker nodes using
% dynamic load balancing.  Every node writes the solutions to file
% when its job is done and sends a message to the manager asking
% for the next job.
```

```
%
tic                                                % start the timer
info = MPI_Init;                                   % MPI startup
[info rank] = MPI_Comm_rank(MPI_COMM_WORLD);
[info nprc] = MPI_Comm_size(MPI_COMM_WORLD);
path(LOADPATH,'/huis/phcpack/PHClab');             % prepare use of PHClab
set_phcpath('/huis/phcpack/PHCv2/bin/phc');
if rank == 0                                       % code for the manager
   origamisys = extract_sys('alignmentequations.txt');
   distribute_tasks(nprc,origamisys);
   fprintf('elapsed time = %.3f s\n',toc);
else
   worker_solves_system();                         % code for the workers
end
info = MPI_Finalize;
LAM_Clean;
quit
end
```

Each origami system described in [1] has 4 inhomogeneous equations in 4 variables and other free parameters. The mixed volume of Newton Polytopes serve as a sharp upper bound for the number of solutions of these origami systems because of the generic parameters. The output of a run on our Rocketcalc cluster configuration with 13 workers is below:

```
[phcpack@idefix Origami]$ mpirun -c 13 octave-2.1.64 -q --funcall origami
Task tallies:
n0   18 (local)
n01   14
n02   14
n03   14
n04   11
n05   12
n06   13
n07   13
n08   12
n09   15
n10   15
n11   17
n12   15
sum  183 (SIZE 183)
elapsed time = 371.603 s
```

6 A Numerical Irreducible Decomposition

There is not (yet) a blackbox solver in PHCpack to compute a numerical irreducible decomposition. In the subsections below we describe the functions which call the tools of `phc`. We start by defining how we represent positive dimensional solution set.

6.1 Witness Sets

To obtain a numerical representation of a positive dimensional solution set, we add as many random hyperplanes as the expected top dimension. Extra slack variables are introduced to turn the augmented system into a square system (i.e.: having as many equations as unknowns) for which we may then apply the blackbox solver.

We illustrate our methods on a special Stewart-Gough platform, which are "architecturally singular" like the so-called Griffis-Duffy platform [4], analyzed in [6]; also see [17]. Once a witness set has been computed, the numerical irreducible decomposition in PHCpack applies monodromy [15] and linear traces [16].

A witness set consists of a polynomial system and a set of solutions which satisfy this system. The polynomial system contains the original polynomial system augmented with hyperplanes whose coefficients are randomly chosen complex numbers. The number of hyperplanes added to the original system equals the dimension of the solution set. The number of solutions in the witness set equals the degree of the solution set.

There are two methods to compute witness sets. The (chronologically) first method is to work top down, starting at the top dimensional solution component and using a cascade [14] of homotopies to compute (super) witness sets as numerical representations of solution sets of all dimensions. The second method works top down, processing equation by equation [19].

6.2 Top Down Computation using a Cascade

The input to **embed** is a system of 8 equations[2] and the number 1, which is the expected top dimension. We solve the embedded system with **solve_system** and then run **cascade** to look for isolated solutions.

```
S = read_system('gdplatB');    % read the system from file
E = embed(S,1);                % embed with one extra hyperplane
sols = solve_system(E);        % call the blackbox solver
size(sols,2)                   % to see candidate witness #points
[sw,R] = cascade(E,sols)       % perform a cascade
```

[2]Maple code to generate the equations is at `http://www.math.uic.edu/~jan/FactorBench/grifdufAe1.html`.

The blackbox solver returns 40 solutions of the embedded system, which turns out the degree of the one dimension curve, because **cascade** finds no other isolated solutions. This can be read from the output shown on screen:

```
ans =

    40

sw =

                []
    [1x40 struct]

R =

             []
    {9x1 cell}
```

The function **cascade** returns two arrays. The first array contains the solutions, while the second one contains the embedded systems. A witness set for a k-dimensional solution is defined by the $(k+1)$-th entries of the arrays returned by **cascade**.

The top down approach has the disadvantage that it requires the user to enter an expected top dimension. While in many practical applications one can guess this top dimension from the context in which the application arises, the default value – taking it as high as the number of variables minus one – is often too expensive.

6.3 Bottom Up Computation: Equation-by-Equation

The new equation-by-equation solver [19] relieves the user from submitting a top dimension and seems more flexible. A disadvantage of the solver is that its performance depends on the order of equations. For the equation describing our Griffis-Duffy platform, we move the simplest equations first.

```
p = read_system('gdplatBa')
[sw,R] = eqnbyeqn(p)

p =

    ' g0*h0+g1*h1+g2*h2+g3*h3'
    ' g0^2+g1^2+g2^2+g3^2-h0^2-h1^2-h2^2-h3^2'
    [1x102 char]
    [1x308 char]
    [1x333 char]
    [1x333 char]
    [1x308 char]
    [1x308 char]

sw =

               []
    [1x40 struct]

R =

            []
    {9x1 cell}
```

6.4 Factoring into Irreducible Components

We continue with the output (`sw`,`R`), computed either with **cascade** or **eqnbyeqn**.

```
dc = decompose(R{2},sw{2,1})     % decompose into irreducible factors

ans =

    40

irreducible factor 1:

ans =

1x28 struct array with fields:
    time
    multiplicity
    err
    rco
    res
    h0
    h1
    h2
    h3
    g3
    g1
    g2
    g0
    zz1

irreducible factor 2:

ans =

            time: 1
    multiplicity: 1
             err: 5.103000000000000e-15
             rco: 1
             res: 3.598000000000000e-15
              h0: -3.091000000000000e-01 - 2.563000000000000e-01i
              h1: -4.439300000000000e-01 + 5.353800000000000e-01i
              h2: 2.563000000000000e-01 - 3.091000000000000e-01i
              h3: -5.353800000000000e-01 - 4.439300000000000e-01i
              g3: 4.766100000000000e-01 + 8.732700000000000e-01i
              g1: 1.164500000000000e+00 - 2.351500000000000e-01i
              g2: -3.360600000000000e-01 + 4.145700000000000e-01i
              g0: -3.643000000000000e-03 + 8.404300000000000e-01i
             zz1: 8.171700000000000e-16 + 5.026400000000000e-16i

......  % 12 more similar linear factors are not shown to save space

dc =

    [1x28 struct]
    [1x1  struct]
    [1x1  struct]
    [1x1  struct]
    [1x1  struct]
    [1x1  struct]
    [1x1  struct]
    [1x1  struct]
    [1x1  struct]
    [1x1  struct]
    [1x1  struct]
    [1x1  struct]
    [1x1  struct]
```

The output of **decompose** shows one irreducible component of degree 28 and 12 lines.

References

[1] R.C. Alperin and R.J. Lang. One and Two-Fold Origami Axioms *2006 4OSME Proceedings*, Pasadena, CA, A.K.Peters, 2007.

[2] J. Fernández, M. Anguita, E. Ros, and J.L. Bernier. SCE Toolboxes for the development of high-level parallel applications. *Proceedings of ICCS 2006*, Volume 3992 of Lecture Notes in Computer Science, pages 518-525, Springer Berlin/Heidelberg, 2006

[3] T. Gao, T.Y. Li, and M. Wu. Algorithm 846: MixedVol: a software package for mixed-volume computation. *ACM Trans. Math. Softw.*, 31(4):555–560, 2005.

[4] M. Griffis and J. Duffy. Method and apparatus for controlling geometrically simple parallel mechanisms with distinctive connections. US Patent 5,179,525, 1993.

[5] B. Huber and B. Sturmfels. A polyhedral method for solving sparse polynomial systems. *Math. Comp.*, 64(212):1541–1555, 1995.

[6] M.L. Husty and A. Karger. Self-motions of Griffis-Duffy type parallel manipulators. *Proc. 2000 IEEE Int. Conf. Robotics and Automation*, CDROM, San Francisco, CA, April 24–28, 2000.

[7] A. Leykin and J. Verschelde. PHCmaple: A Maple interface to the numerical homotopy algorithms in PHCpack. In Quoc-Nam Tran, editor, *Proceedings of the Tenth International Conference on Applications of Computer Algebra (ACA'2004)*, pages 139–147, 2004.

[8] A. Leykin and J. Verschelde. Interfacing with the numerical homotopy algorithms in PHCpack. In Nobuki Takayama and Andres Iglesias, editors, *Proceedings of ICMS 2006*. Volume 4151 of Lecture Notes in Computer Science, pages 354–360, Springer-Verlag, 2006.

[9] A. Leykin, J. Verschelde, and A. Zhao. Newton's method with deflation for isolated singularities of polynomial systems. *Theoretical Computer Science* 359(1-3): 111-122, 2006.

[10] T.Y. Li. Numerical solution of polynomial systems by homotopy continuation methods. In F. Cucker, editor, *Handbook of Numerical Analysis. Volume XI. Special Volume: Foundations of Computational Mathematics*, pages 209–304. North-Holland, 2003.

[11] M. Maekawa, M. Noro, K. Ohara, Y. Okutani, N. Takayama, and Y. Tamura. OpenXM – an open system to integrate mathematical softwares. Available at `http://www.OpenXM.org/`.

[12] M. Maekawa, M. Noro, K. Ohara, N. Takayama, and Y. Tamura. The design and implementation of OpenXM-RFC 100 and 101. In K. Shirayanagi and K. Yokoyama, editors, *Computer mathematics. Proceedings of the Fifth Asian Symposium (ASCM 2001) Matsuyama, Japan 26 - 28 September 2001*, volume 9 of *Lecture Notes Series on Computing*, pages 102–111. World Scientific, 2001. Available at `http://www.math.kobe-u.ac.jp/OpenXM/ascm2001/ascm2001/ascm2001.html.`

[13] J.M. McCarthy. *Geometric Design of Linkages.* Volume 11 of Interdisciplinary Applied Mathematics, Springer-Verlag, 2000.

[14] A.J. Sommese and J. Verschelde. Numerical homotopies to compute generic points on positive dimensional algebraic sets. *J. of Complexity*, 16(3):572–602, 2000.

[15] A.J. Sommese, J. Verschelde, and C.W. Wampler. Using monodromy to decompose solution sets of polynomial systems into irreducible components. In C. Ciliberto, F. Hirzebruch, R. Miranda, and M. Teicher, editors, *Application of Algebraic Geometry to Coding Theory, Physics and Computation*, pages 297–315. Kluwer Academic Publishers, 2001. Proceedings of a NATO Conference, February 25 - March 1, 2001, Eilat, Israel.

[16] A.J. Sommese, J. Verschelde, and C.W. Wampler. Symmetric functions applied to decomposing solution sets of polynomial systems. *SIAM J. Numer. Anal.*, 40(6):2026–2046, 2002.

[17] A.J. Sommese, J. Verschelde, and C.W. Wampler. Advances in polynomial continuation for solving problems in kinematics. *ASME Journal of Mechanical Design* 126(2):262–268, 2004.

[18] A.J. Sommese, J. Verschelde, and C.W. Wampler. Numerical irreducible decomposition using PHCpack. In M. Joswig and N. Takayama, editors, *Algebra, Geometry, and Software Systems*, pages 109–130. Springer–Verlag, 2003.

[19] A.J. Sommese, J. Verschelde, and C.W. Wampler. Solving polynomial systems equation by equation. Submitted for publication.

[20] A.J. Sommese and C.W. Wampler. *The Numerical solution of systems of polynomials arising in engineering and science.* World Scientific Press, Singapore, 2005.

[21] J. Verschelde. Algorithm 795: PHCpack: A general-purpose solver for polynomial systems by homotopy continuation. *ACM Trans. Math. Softw.*, 25(2):251–276, 1999. Software available at `http://www.math.uic.edu/~jan`.

Appendix A: Alphabetic List of PHClab Functions

Below is an alphabetic list of the functions offered by PHClab.

cascade executes a sequence of homotopies, starting at the top dimensional solution set to find super witness sets. The input consists of an embedded system (the output of **embed**) and its solutions (typically obtained via **solve_system**. The output of this function is a sequence of super witness sets. A witness set is a numerical representation for a positive dimensional solution set. The “super” means that the k-th super witness set may have junk points on solutions sets of dimension higher than k.

decompose takes a witness set on input and decomposes it into irreducible factors. The witness set is represented by two input parameters: an embedded system and solutions which satisfy it. The number of solutions equal the degree of the pure dimensional solution set represented by the witness set. On return is a sequence of witness sets, each witness set in the sequence corresponds to one irreducible component.

deflation reconditions isolated singular solutions. The input consists of two parameters: a polynomial system and a sequence of approximate solutions to the system. Typically these solutions are obtained via the blackbox solver or as the output of the function **track**. On return is a list of reconditioned solutions, along with the augmented systems which have as regular solution the multiple solution of the original system.

embed adds extra hyperplanes and slack variables to a system, as many as the expected top dimension of the solution set. There are two input parameters: a polynomial system and the number of hyperplanes which have to be added. Typically, this number is the top dimension of the solution set. If nothing is known about this top dimension, a default value for this number is the number of variables minus one.

eqnbyeqn solves polynomial systems equation by equation. For the polynomial system on input, this function returns a sequence of witness sets. The kth witness set in the sequence is a numerical representation of the solution set of dimension k.

make_system converts the matrix format of a system into a symbolic format acceptable to `phc`. A polynomial system of N equations in n variables, with a total of m terms, is represented by a matrix with $N + m$ rows and $n + 1$ columns. Each polynomial is terminated by a zero row in the matrix. Each row represents one term in a polynomial, starting with its (complex) coefficient and continuing with the values of the exponents for each variable.

mixed_volume computes the mixed volume for a system of n equations in n variables. There are two input parameters: the system and a flag to indicate whether a random coefficient system must be created and solved. If the flag on input is one, then on return is a start system which has as many solutions as the mixed volume.

phc_filter removes from a super witness set those junk points while lie on a higher dimensional solution set. The third and last input parameter is a set of points to be filtered. The first two parameters represent a witness set, given by an embedded system and a sequence of solutions which satisfy the embedded system. On return are those points of the third input that do not lie on the component represented by the witness set.

read_system reads a polynomial system from file. There is only one input parameter: a file name. The format of the polynomial system on file must follow the input format of PHCpack. The function returns an array of strings, each string in the array is a multivariate polynomial in symbolic format.

refine_sols applies Newton's method to refine a solution. There are six input parameters: a polynomial system, an approximate solution, a tolerance for the magnitude of the correction vector `err`, a threshold for to decide whether a solution is singular (relative to `rco`), a tolerance for the residual `res`, and finally a natural number with the maximal number of Newton iterations that are allowed. On return is an array of refined approximate solutions.

set_phcpath defines the directory where the executable version of phc is. For example, if the program `phc` is in the directory `/tmp`, then **set_phcpath**('/tmp/phc') must be executed at the start of a PHClab session. On Windows, `'/tmp/phc'` could be replaced by `'C:/Downloads/phc'` if `phc.exe` is in the directory `Downloads` on the C drive.

solve_system calls the blackbox solver of phc. On input is a polynomial system in matrix format, see the input description for the command **make_system**. An alternative input format is the cell array returned by **read_system**. The output is an array of structures. Every element in the array contains one solution at the end of a solution path. In addition to the values for the coordinates of the solution, an estimate for the condition number of the solution which leads to a measure for the forward error, while the residual measures the backward error.

track applies numerical continuation methods for a homotopy between start and target system, for a specified set of start solutions. The three arguments for **track** are respectively the target system, the start system and the solutions of the start system. The target and start system must be given in matrix format. If the start solutions are singular, then the path tracker will fail to start. The output of **track** is an array of the same length as the array of start solutions, containing the values at the end of the solution paths.

Appendix B: Exercises

1. Use the blackbox solver to solve (the `phc` input format is on the right):

$$\left\{ \begin{array}{l} x^2 + y^2 - 1 = 0 \\ x^3 + y^3 - 1 = 0 \end{array} \right. \qquad \begin{array}{l} \texttt{2} \\ \texttt{ x\^{}2 + y\^{}2 - 1;} \\ \texttt{ x\^{}3 + y\^{}3 - 1;} \end{array} \tag{3}$$

   The exact solutions are (1,0), (0,1), $(-1 + i\frac{\sqrt{2}}{2}, -1 - i\frac{\sqrt{2}}{2})$ and $(-1 - i\frac{\sqrt{2}}{2}, -1 + i\frac{\sqrt{2}}{2})$. How many solutions does **solve_system** return? Verify whether the output matches the exact solutions. Use **refine_sols** to discover what the multiplicities of the solutions (0,1) and (1,0) are.

2. If we have to solve repeatedly a polynomial system with the same structure, we may want to save a start system. To solve systems with the same monomials as in (3), we could use

$$g(x, y) = \left\{ \begin{array}{l} x^2 + 1.232y^2 + 1.1211i = 0 \\ y^3 - 0.872y^2 - 0.6231 + 1.032i = 0 \end{array} \right. \tag{4}$$

   Since the coefficients are random complex numbers (feel free to make other <u>*random*</u> choices) all solutions of the system $g(x, y) = \mathbf{0}$ will be regular.

   (a) Solve the system $g(x, y) = \mathbf{0}$, using **solve_system**. Verify that all solutions are regular.

   (b) Use **track** to solve the system in (3).
   Check whether you find the same solutions, eventually computed in a different order.

3. The following system has multiple roots:

$$\left\{ \begin{array}{l} x^2 + y - 3 = 0 \\ x + 0.125y^2 - 1.5 = 0 \end{array} \right. \tag{5}$$

   (a) Use **solve_system** to find approximate roots. Can you see which roots are multiple?

   (b) Apply **deflation** to the approximate roots.
   Observe the values of the field `rco` of the solutions before and after the deflation.

   (c) What is the multiplicity of each solution?

4. All adjacent minors of an indeterminate 2-by-4 matrix for a system of 3 equations in 8 variables:

$$\left\{ \begin{array}{l} x_{11}x_{22} - x_{21}x_{12} = 0 \\ x_{12}x_{23} - x_{22}x_{13} = 0 \\ x_{13}x_{24} - x_{23}x_{14} = 0. \end{array} \right. \tag{6}$$

   (a) Use **embed** to add 5 random hyperplanes.

   (b) Solve the embedded system. What is the degree of this 5-dimensional solution set?

   (c) Apply **decompose** to factor the solution set. How many irreducible factors do you find?

   (d) Repeat the process for larger instances of this problem, for $n = 5, 6, \ldots$.

5. Consider the system

$$\begin{array}{rcccl} (x_1^2 - x_2)(x_1 - 0.5) & = & x_1^3 - 0.5x_1^2 - x1x_2 + 0.5x_2 & = & 0 \\ (x_1^3 - x_3)(x_2 - 0.5) & = & x_1^3 x_2 - 0.5x_1^3 - x_2x_3 + 0.5x_3 & = & 0 \\ (x_1x_2 - x_3)(x_3 - 0.5) & = & x_1x_2x_3 - 0.5x_1x_2 - x_3^2 + 0.5x_3 & = & 0. \end{array} \tag{7}$$

   Solving this system means to compute witness sets for all irreducible factors.

   (a) Use **embed** to add 1 random hyperplane.

   (b) Solve the embedded system with **solve_system**.
   Among the solutions, can you see the three witness points on the twisted cubic? Look for solutions with a slack variable close to zero.

   (c) Apply **cascade** to find candidate isolated solutions.

   (d) Use **phc_filter** to filter the candidate isolated solutions.
   How many isolated solutions does the system have?