Max-Planck-Institut für Mathematik in den Naturwissenschaften Leipzig

HomotopyContinuation.jl - a package for solving systems of polynomial equations in Julia

by

Paul Breiding and Sascha Timme

Preprint no.: 1 2018

HomotopyContinuation.jl

Paul Breiding* Sascha Timme†

Abstract

This paper is gives a short introduction into HomotopyContinuation.jl, a polynomial equations solver for Julia [4]. It present excerpts from the comprehensive documentation available at https://juliahomotopycontinuation.github.io/latest/.

The aim of this project is twofold: establishing a fast numerical polynomial solver in Julia and at the same time providing a highly customizable algorithmic environment for researchers for designing and performing individual experiments.

Contents

*Max-Planck-Institute for Mathematics in the Sciences Leipzig (breiding-at-mis.mpg.de)

†TU Berlin (sascha.timme-at-googlemail.com)

1 Getting started

HomotopyContinuation.jl is a package for solving square polynomial systems via homotopy continuation in Julia. It is available under

https://github.com/JuliaHomotopyContinuation/HomotopyContinuation.jl

or, alternatively, you can install it in Julia using the command

```
Pkg.add("HomotopyContinuation");
```

We recommend using the Atom editor[1] and its uber−juno package. You need to have installed at least Julia v. 0.6.1.

HomotopyContinuation.jl aims at having easy-to-understand top-level commands. For instance, suppose we wanted to solve the following system

$$f = \begin{bmatrix} x^2 + y \\ y^2 - 1 \end{bmatrix}.$$

First, we have to define f in Julia. For human-readable and easy constructable input HomotopyContinuation.jl is using the MultivariatePolynomials.jl[2] interface. In the following we will use its DynamicPolynomials.jl[3] implementation.

```
import DynamicPolynomials: @polyvar
# @polyvar is a function for initializing variables.

@polyvar x y # initialize the variables x y
f = [x^2+y, y^2-1]
```

To solve $f = 0$ we execute the following command.

```
using HomotopyContinuation # load the module HomotopyContinuation
solve(f) # solves for f=0
```

(see Section 4.1 for a list of options that can be passed to 'solve').

The last command will return a type HomotopyContinuation.Result{Complex{Float64}} of length 4 (one entry for each solution):

```
julia> ans

julia> HomotopyContinuation.Result{Complex{Float64}}
    * paths -> 4
    * successfull paths -> 4
    * solutions at infinity -> 0
    * singular solutions -> 0
    * real solutions -> 2
    HomotopyContinuation.PathResult{Complex{Float64}}[4]
```

Let us see what is the information that we get. Four paths were attempted to be solved, four of which were completed successfully. Since we tried to solve an affine system, the algorithm checks whether there are solutions at infinity: in this case there are none. None of the solutions is singular and two of them are real. To access the first solution in the array we write

[1] https://www.atom.io
[2] https://github.com/JuliaAlgebra/MultivariatePolynomials.jl
[3] https://github.com/JuliaAlgebra/DynamicPolynomials.jl

```
julia> ans[1]

julia> HomotopyContinuation.PathResult{Complex{Float64}}
    returncode -> :success
    solution -> Complex{Float64}[2]
    singular -> false
    residual -> 1.02e-15
    newton_residual -> 8.95e-16
    log10_condition_number -> 0.133
    windingnumber -> 1
    angle_to_infinity -> 0.615
    real_solution -> true
    startvalue -> Complex{Float64}[2]
    iterations -> 17
    endgame_iterations -> 5
    npredictions -> 2
    predictions -> Vector{Complex{Float64}}[2]
```

The returncode tells us that the pathtracking was successfull. What do the other entries of the table tell us? Let us consider the most relevant (for a complete list of explanations consider Section 4.2).

- solution: the zero that is computed (here it is $[-1, -1]$).
- singular: boolean that shows whether the zero is singular.
- residual: the computed value of $|f([-1, -1])|$.
- angle_to_infinity: the algorithms homogenizes the system f and then computes all solutions in projective space. The angle to infinity is the angle of the solution to the hyperplane where the homogenizing coordinate is 0.
- real_solution: boolean that shows whether the zero is real.

Suppose we were only interested in the real solutions. The command to extract them is

```
solutions(solve(f), only_real=true)
```

(a detailed explanation of the 'solutions' function is in Section 4.3). Indeed, we have

```
julia> [ans[i].solution for i=1:2]
julia> Vector{Complex{Float64}}[2]
    Complex{Float64}[2]
    1.00 - 2.66e-15 im
    -1.00 + 1.33e-15 im
    Complex{Float64}[2]
    -1.00 + 2.72e-15 im
    -1.00 + 1.44e-15 im
```

which are the two real zeros of f. By assigning the boolean values in the solutions function we can filter the solutions given by solve(f) according to our needs.

We solve some more elaborate systems in Section 2. HomotopyContinuation.jl also supports input of type BigFloat.

2 Examples

2.1 Computing the degree of a variety

Consider the projective variety in the 2-dimensional complex projective space $\mathbb{CP}^2$.

$$V = \{x^2 + y^2 - z^2 = 0\}$$

The degree of V is the number of intersection points of V with a generic line. Let us see what it is. First we initialize the defining equation of V.

```
import DynamicPolynomials: @polyvar

@polyvar x y z
f = x^2 + y^2 - z^2
```

Let us sample the equation of a random line.

```
L = randn(1,3) * [x; y; z]
```

Now we compute the number of solutions to $[f = 0, L = 0]$.

```
using HomotopyContinuation
solve([f; L])
```

We find two distinct solutions and conclude that the degree of V is 2.

2.2 The angles in a triangle

The following example is from [1, Section 7.3.].

Consider a triangle with sides a, b, c and let θ be the angle opposite of c. The goal is to compute θ from a, b, c. We define $s_\theta := \sin\theta$ and $c_\theta := \cos\theta$. The corresponding polynomial system is.

```
import DynamicPolynomials: @polyvar
a = 5
b = 4
c = 3
@polyvar s_theta c_theta
f = [c_theta^2 + s_theta^2 - 1,
    (a * c_theta - b)^2 + (a * s_theta)^2 - c^2]
```

To set up a totaldegree homotopy of type StraightLineHomotopy (see Section 3.2.3) we have to write

```
using HomotopyContinuation
H, s = totaldegree(StraightLineHomotopy, f)
```

This sets up a homotopy H of the specified type using a random starting system that comes with a vector s of solutions. To solve for $f = 0$ we execute

```
solve(H, s)
```

If instead we wanted to use GeodesicOnTheSphere (see Section 3.2.2) as homotopy type, we write

```
H, s = totaldegree(GeodesicOnTheSphere, f)
solve(H, s)
```

The angles are of course only the real solutions of $f = 0$. We get them by using

```
solution(ans, only_real=true)
```

2.3 Binding polynomials

The following polynomial system is the example from Section 5.1 in [3]. It is called a *binding polynomial*.

```
using HomotopyContinuation
import DynamicPolynomials: @polyvar

@polyvar w[1:6]

f = [
    11*(2*w[1]+3*w[3]+5*w[5])+13*(2*w[2]+3*w[4]+5*w[6]),
    11*(6*w[1]*w[3]+10*w[1]*w[5]+15*w[3]*w[5])
        +13*(6*w[2]*w[4]+10*w[2]*w[6]+15*w[4]*w[6]),
    330*w[1]*w[3]*w[5]+390*w[2]*w[4]*w[6],
    143*(2*w[1]*w[2]+3*w[3]*w[4]+5*w[5]*w[6]),
    143*(6*w[1]*w[2]*w[3]*w[4]+10*w[1]*w[2]*w[5]*w[6]
            +15*w[3]*w[4]*w[5]*w[6]),
    4290*w[1]*w[2]*w[3]*w[4]*w[5]*w[6]
    ];
```

Suppose we wanted to solve $f(w) = a$, where

```
a=[71, 73, 79, 101, 103, 107]
```

To get an initial solution we compute a random forward solution.

```
w_0 = randn(6)
a_0 = map(p -> p(w => w_0), f)
```

Now we set up the homotopy.

```
H = StraightLineHomotopy(f-a_0, f-a)
```

and compute a backward solution with starting value w_0 by

```
solve(H, w_0)
```

By default the solve function uses SphericalPredictorCorrector as the pathtracking routing. To use the AffinePredictorCorrector instead we must write

```
solve(H, w_0, AffinePredictorCorrector())
```

The system $f = 0$ has 72 simple non-real roots. The command

```
S = solve(f-a);
solutions(S, singular = false)
```

however, only returns 62. The reason is that the remaining 10 solutions are ill-conditioned. We find all 72 solutions by

```
S = solve(f-a, singular_tol=1e8);
solutions(S, singular = false)
```

The default of singular_tol in HomotopyContinuation.jl is 10^4.

2.4 6-R Serial-Link Robots

The following example is from [1, Section 9.4].

Consider a robot that consists of 7 links connected by 6 joints. The first link is fixed on the ground and the last link has a "hand". The problem of determining the position of the hand when knowing the arrangement of the joints is called *forward problem*. The problem of determining any arrangement of joints that realized a fixed position of the hand is called *backward problem*. Let us denote by $z_1, \dots, z_6$ the unit vectors that point in the direction of the joint axes. They satisfy the following polynomial equations

$$\begin{aligned} z_i \cdot z_i &= 1, & i &= 1, \dots, 6 \\ z_i \cdot z_{i+1} &= \cos \alpha_i, & i &= 1, \dots, 5 \\ \sum_{i=1}^{5} a_i \, z_i \times z_{i+1} + \sum_{i=2}^{5} a_{i+4} \, z_i &= p \end{aligned}$$

for some (α, a) and a known p (see [1] for a detailed explanation on how these numbers are to be interpreted).

In this notation the forward problem consists of computing (α, a) given the z_i and p and the backward problem consists of computing $z_2, \dots, z_5$ that realize some fixed (α, a, z_1, z_6) (knowing z_1, z_6 means that the position where the robot is attached to the ground and the position where its hand should be are fixed).

We now compute first a forward solution (α_0, a_0), and then use (α_0, a_0) to compute a solution for the backward problem imposed by some random (α, a).

```
using HomotopyContinuation
import DynamicPolynomials: @polyvar

@polyvar z2[1:3] z3[1:3] z4[1:3] z5[1:3]
z1 = [1, 0, 0]
z6 = [1, 0, 0]
p = [1, 1, 0]
z = [z1, z2, z3, z4, z5, z6]

f = [dot(z[i], z[i]) for i=2:5]
g = [dot(z[i], z[i+1]) for i=1:5]
h = hcat([[cross(z[i], z[i+1]) for i=1:5]; [z[i] for i=2:5]]...)

alpha = randexp(5)
a = randexp(9)
```

Let us compute a random forward solution by sampling an assignment for $z_2, \dots, z_5$.

```
z_0=rand(3,4); # Compute a random assignment for the variable z
for i = 1:4
    # normalize the columns of z_0 to norm 1
    z_0[:,i] = z_0[:,i] ./ norm(z_0[:,i])
end
# vectorize z_0,
z_0 = vec(z_0)
```

from z_0 we compute a forward solution.

```
# compute the forward solution of alpha
alpha_0 = map(p -> acos(p([z2; z3; z4; z5] => z_0)), g)

# evaluate h at z_0
h_0 = map(p -> p([z2; z3; z4; z5] => z_0), h)
# compute a solution to h(z_0) * a = p
a_0 = h_0\p
```

Using the forward solution (α_0, a_0) we construct the following StraightLineHomotopy.

```
F_1 = [f-1; g-cos.(alpha_0); h*a_0-p]
F_2 = [f-1; g-cos.(alpha); h*a-p]
H = StraightLineHomotopy(F_1, F_2)
```

To compute a backward solution with starting value z_0 we finally execute

```
solve(H, z_0)
```

To compute all the backward solutions we may perform a totaldegree homotopy. Although the Bezout number of the system is 1024 the generic number of solutions is 16. We find all 16 solutions by

```
H, s = totaldegree(StraightLineHomotopy, F_1)
solutions(solve(H, s), singular=false)
```

On a MacBook Pro with 2,6 GHz Intel Core i7 and 16 GB RAM memory the above operation takes about 572 seconds. With parallel computing provided by the addprocs() command in Julia it takes about 93 seconds.

2.5 Random homotopies

The command

```
randomsystem(5, 2, mindegree=1, maxdegree=2, density=0.5)
```

creates a random polynomial system with 5 equations in 2 variables named x_1, x_2. Each polynomial has a total degree drawn uniformly from $[\text{mindegree}, \ldots, \text{maxdegree}] = \{1, 2\}$. The value of density determines the fraction of monomials having a non-zero coefficient.

The randomsystem() command is built into another function that sets up a random homotopy:

```
randomhomotopy(StraightLineHomotopy, 2)
```

creates a total degree homotopy (see Section 3.2.3) where the target system is consists of two equations in two variables created with randomsystem(). Example:

```
julia> H, solutions = randomhomotopy(StraightLineHomotopy,
                                     2, mindegree=3, maxdegree=6);
julia> length(H)
2
julia> nvariables(H)
2
```

We want to use randomhomotopy() to see how good HomotopyContinuation.jl copes with sparse system compared to dense systems. For each $i \in \{0.3, 0.6, 1\}$ we construct 100 random homotopies with 3 equations in three variables, all of degree 4 and with density i. Then, we solve the homotopy and record the time in seconds. We first initialize an array to record the times.

```
times = zeros(100,3)
d = [0.3  0.6  1.0]
```

Then, we make the experiment.

```
using HomotopyContinuation
    for i =1:3
        for k = 1:100
            H,s = randomhomotopy(StraightLineHomotopy{Complex128}, 3,
                                 mindegree = 4, maxdegree = 4,
                                 density = d[i])
            tic()
            solve(H,s)
            times[k,i] = toc()
        end
    end
```

The mean values of the colums of times are

```
julia> mean(times, 1)
1x3 Array{Float64,2}:
 0.205056  0.118019  0.140468
```

As expected a density of 0.3 takes longest. It is surprising though that a density of 0.6 on the average was computed faster than a density of 1.0. The experiment was made on a MacBook Pro with 2,6 GHz Intel Core i7 and 16 GB RAM memory.

3 Homotopies

Homotopies.jl is a separate package for constructing (polynomial) homotopies $H(x,t)$. We export in HomotopyContinuation every function from Homotopies.

Each homotopy has the same interface so that you can switch easily between different homotopy types. Based on this interface there are also provided some convenient higher level constructs; e.g., the construction of a total degree system and its start solutions. Homotopies.jl comes with an an interface to MultivariatePolynomials.jl and its DynamicPolynomials.jl implementation for human-readable input and output. Internally, Homotopies.jl uses the package FixedPolynomials.jl for fast evaluation.

3.1 Setting up a homotopy – an example

As an example we construct a homotopy between the polynomial systems

$$f = \begin{bmatrix} x + y^3 \\ x^2 y - 2y \end{bmatrix}, \quad g = \begin{bmatrix} x^3 + 2 \\ y^3 + 2 \end{bmatrix}.$$

Currently, there are two types of homotopies implemented:

```
StraightLineHomotopy
GeodesicOnTheSphere
```

see Section 3.2. Suppose we wanted to construct a homotopy between the following systems.

```
import DynamicPolynomials: @polyvar
@polyvar x y # initilize the variables x y

f = [x + y^3, x^2*y-2y]
g = [x^3+2, y^3+2]
```

The code to initialize a StraightLineHomotopy is as follows.

```
using HomotopyContinuation

H = StraightLineHomotopy(f, g)
# H is now StraightLineHomotopy{Int64}

# to avoid unnecessary conversions one could also have
H = StraightLineHomotopy{Complex128}(f, g)

# we can now evaluate H
evaluate(H, rand(Complex128, 2), 0.42)
# or alternatively
H(rand(Complex128, 2), 0.42)
```

3.2 Homotopies

The following homotopies are implemented. They are subtypes of AbstractPolynomialHomotopy.

3.2.1 StraightLineHomotopy

With the command

```
StraightLineHomotopy(f, g)
```

we construct the homotopy

$$H(x,t) = tf(x) + (1-t)g(x)$$

(f is the start-system). The systems f and g have to match and to be one of the following

- Vector{<:MP.AbstractPolynomial}
- {MP.AbstractPolynomial}
- {Vector<:FP.Polynomial},

where 'MP' is 'MultivariatePolynomials' and 'FP' is 'FixedPolynomials'. You can also force a specific coefficient type 'T by typing

```
StraightLineHomotopy{T}(f, g)
```

3.2.2 GeodesicOnTheSphere

With the command

```
GeodesicOnTheSphere(f, g)
```

we construct the homotopy

$$H(x,t) = \frac{\sin(t\alpha)}{\sin(\alpha)} f + \left(\cos(t\alpha) - \frac{\sin(t\alpha)\cos(\alpha)}{\sin(\alpha)}\right) g$$

(f is the start-system). I.e., $H(x,t)$ is the geodesic on the sphere from f to g. The systems f and g have to match and to be one of the following

- Vector{<:MP.AbstractPolynomial}
- {MP.AbstractPolynomial}
- {Vector<:FP.Polynomial},

where 'MP' is 'MultivariatePolynomials' and 'FP' is 'FixedPolynomials'. You can also force a specific coefficient type 'T by typing

```
GeodesicOnTheSphere{T}(f, g)
```

3.2.3 Total degree homotopy

The command

```
totaldegree(H::Type{AbstractPolynomialHomotopy}, F, [unitroots=false])
```

constructs a total degree homotopy of type 'H' with F and the start system

$$\begin{aligned} z_1^{d_1} - b_1 \\ z_1^{d_2} - b_2 \\ \vdots \\ z_n^{d_n} - b_n \end{aligned}$$

together with all its solutions. Here d_i is the degree of the i-th polynomial of F. If unitroots=**true**, then $b_i = 1$. Otherwise b_i is a random complex number (with real and imaginary part in the unit interval). Example:

```
H, startsolutions = totaldegree(StraightLineHomotopy, [x^2-1, y-2])
```

3.3 Condition numbers

Homotopies.jl provides three notions of a condition number for polynomial solving from [2]. The first is from [2, Proposition 16.10].

$$\kappa(f,x) := \|f\| \, \|Df(x)^\dagger \mathrm{diag}(\|x\|^{d_i - 1})\|.$$

can be called by

```
kappa(H, x, t),
```

where $H(x,t) = f(x)$; i.e., the function kappa is defined with respet to homotopies and not polynomials. The d_i are the the degrees of the polynomials in f.

The second it the normalized version of κ from [2, Eq. (16.11)] and it is defined as

$$\kappa_{\text{norm}}(f,x) := \|f\| \, \|Df(x)^\dagger \text{diag}(\sqrt{d_i}\, \|x\|^{d_i-1})\|.$$

It is called by

```
kappa_norm(H, x, t).
```

The third is the μ_{norm} condition number from [2, Definition 16.43]:

$$\mu_{\text{norm}}(f,x) := \|f\| \, \|Df(x)|_{x^\perp}^{-1} \text{diag}(\sqrt{d_i}\, \|x\|^{d_i-1})\|.$$

It is called by writing

```
mu_norm(H, x, t).
```

HomotopyContinuation.jl uses $\kappa(f,x)$ to decide for ill-posedness.

4 Solving homotopies

To solve the homotopy H with the given start values startvalues_s type

```
solve(H::AbstractHomotopy, startvalues_s)
```

The function also takes polynomials as inputs:

```
solve(f::Vector{<:MP.AbstractPolynomial{T}})
```

4.1 Solver options

The solve function supports the following options:

- endgame_start=0.1: Where the endgame starts.
- abstol=1e−12 The desired accuracy of the final roots.
- at_infinity_tol=1e−10: An point is at infinity if the maginitude of the homogenous variable. is less than at_infinity_tol.
- singular_tol=1e4: If the winding number is 1 but the condition number is larger than singular_tol then the root is declared being singular.
- refinement_maxiters=100: The maximal number of Newton iterations to achieve abstol.
- verbose=**false**: Print additional warnings / informations.
- apply_gammatrick=**true**: This modifies the start system to by mutiplying it with a random number γ.

- gamma=apply_gammatrick ? exp(im*2*pi*rand()) : complex(1.0): You can overwrite the default gamma. This is useful if you want to rerun only some paths.
- pathcrossing_tolerance=1e−8: The tolerance for when two paths are considered to be crossed.
- pathcrossing_check=**true**: Enable the pathcrossing check.
- parallel_type=:pmap: Currently there are two modes: :pmap will use pmap for parallel computing. and :none will use the standard map. :pmap is enabled by default since it works reliable, but if you develop new algorithms you probably want to disable it. *
- batch_size=1: The batch_size for pmap if parallel_type is :pmap.

For instance, to solve the homotopy H with starting values s with no endgame and a singular tolerance of 1e5, write

```
solve(H, s, endgame_start=0.0, singular_tol=1e5)
```

To solve the polynomial system f with the same options write

```
solve(f, endgame_start=0.0, singular_tol=1e5)
```

4.2 Results

The solve function returns an array of results. Here is an explanation what they mean.

- returncode: One of :success, :at_infinity or any error code from the EndgamerResult.
- solution: The solution vector. For homogeneous systemsor solutions is at infinity the projective solution is returned. Otherwise an affine solution is given if the startvalue was affine and a projective solution is given if the startvalue was projective.
- residual: The value of the infinity norm of $H(\text{solution}, 0)$.
- newton_residual: The value of te 2-norm of $DH(\text{solution}, 0)^{-1}H(\text{solution}, 0)$, where D is the derivative with respect to x.
- log10_condition_number: The value is the logarithm (with base 10) of the condition number.
- windingnumber: The estimated winding number
- angle_to_infinity: The angle to infinity is the angle of the solution to the hyperplane where the homogenizing coordinate is 0.
- real_solution: Indicates whether the solution is real given the defined tolerance at_infinity_tol (from the solver options).
- startvalue: The startvalue of the path.
- iterations: The number of iterations the pathtracker needed.
- endgame_iterations: The number of steps in the geometric series the endgamer did.
- npredictions: The number of predictions the endgamer did.
- predictions: The predictions of the endgamer.

4.3 The solutions function

The solution function filters the solutions satisfing the constraints. The full flag is

```
solutions(r::Result; success=true, at_infnity=true,
                     only_real=false, singular=true)
```

References

[1] A. Sommese, C. Wampler: *The Numerical Solution of Systems of Polynomial Arising in Engineering and Science*, World Scientific (2005).

[2] P. Bürgisser, F. Cucker: *Condition – The geometry of numerical algorithms*, Springer (2013).

[3] Y. Ren, J. W. R. Martini, J. Torres: *Decoupled molecules with binding polynomials of bidegree (n,2)*, ArXiv e-prints 1711.06865.

[4] J. Bezanson, A. Edelman, S. Karpinski and V. Shah: *Julia: A fresh approach to numerical computing*, SIAM Review, 59(1) (2017), 65-98.