Homotopy Continuation!

Miles Cochran-Branson

Friday, November 12

Contents

1 Implementing Homotopy Continuation in Julia

Say we're interested in solving the system of equations

$$f_1(x,y) = (x^4 + y^4 1)(x^2 + y^2 2) + x^5 y$$
$$f_2(x,y) = x^2 + 2xy^2 - 2y^2 - \frac{1}{2}.$$

This can very simply be done using hometopy continuation and the package `HometopyContinuation.jl` in `Julia` as follows

[1]:
```
# load packages
using HomotopyContinuation
```

[2]:
```
# declare variables x and y
@var x y
# define the polynomials
f = (x^4 + y^4 - 1) * (x^2 + y^2 - 2) + x^5 * y
f = x^2 + 2*x*y^2 - 2*y^2 - 1/2
F = System([f, f])
result = solve(F)
```

```
Tracking 18 paths... 100%|| Time:
0:00:26
  # paths tracked:                  18
  # non-singular solutions (real):  18 (4)
```

```
# singular endpoints (real):      0 (0)
# total solutions (real):         18 (4)
```

[2]:
```
Result with 18 solutions
========================
• 18 paths tracked
• 18 non-singular solutions (4 real)
• random_seed: 0x6a9bef16
• start_system: :polyhedral
```

and we can display the solutions

[3]:
```
solutions(result)
```

```
18-element Vector{Vector{ComplexF64}}:
 [0.9443571312488813 + 0.3118635017972661im, 0.32083818529039954 + 0.
 ↪9677296009728283im]
 [0.8999179208471728 - 8.474091755303838e-33im, -1.2441827613422727 + 9.
 ↪244463733058732e-33im]
 [1.0866676911062136 - 0.3290700669978769im, -0.24048106708661118 + 1.
 ↪1350215823993672im]
 [0.06944255588971958 - 1.0734210145259255im, 0.2852544211796043 + 0.
 ↪7076856100161802im]
 [0.9443571312488815 - 0.311863501797266im, 0.3208381852903996 - 0.
 ↪9677296009728285im]
 [0.8708494909007279 + 0.03055030123533535im, 0.9716347867303131 + 0.
 ↪21528324968394982im]
 [-1.0665536440076708 + 0.1424944286215267im, -0.3998429331325364 + 0.
 ↪07871238147493467im]
 [-1.671421392838003 + 0.0im, 0.6552051858720408 + 7.52316384526264e-37im]
 [1.7132941375582666 - 0.5813863945698412im, 0.047514134780854124 + 1.
 ↪252792951007688im]
 [1.0866676911062132 + 0.3290700669978767im, -0.24048106708661127 - 1.
 ↪1350215823993668im]
 [0.8708494909007278 - 0.03055030123533537im, 0.9716347867303131 - 0.
 ↪21528324968394498im]
 [0.8209788924342627 - 1.88079096131566e-36im, -0.6971326459489459 - 6.
 ↪018531076210112e-36im]
 [0.06944255588971956 + 1.0734210145259255im, 0.28525442117960437 - 0.
 ↪7076856100161802im]
 [0.07565391048031057 + 0.9487419814734106im, -0.24800445792173503 + 0.
 ↪6838307098593375im]
 [-0.9368979667963299 + 3.5264830524668625e-38im, 0.312284081738601 + 2.
 ↪82118644197349e-37im]
 [-1.0665536440076708 - 0.1424944286215266im, -0.3998429331325363 - 0.
 ↪07871238147493461m]
```

```
[0.07565391048031057 - 0.9487419814734106im, -0.24800445792173503 - 0.
↪6838307098593376im]
[1.7132941375582666 + 0.5813863945698413im, 0.04751413478085409 - 1.
↪2527929510076 88im]
```

1.1 Implementing our own method

In order to better understand this process we implement our own method for solving. This involves first taking an Euler step, that is, essentially solving a system of ODEs using the Euler method, and then correcting our quasi-solution by using the Newton method.

The implementation starts by fixing a *Homotopy*, $H(x,t)$ which can be expressed in the form

$$H(x,t) = t \cdot f(x) + (1-t) \cdot g(x)$$

where $t \in [0,1] \subset \mathbb{R}$, $f(x)$ is our original system, and $g(x)$ is a simpler system of equations which we *know* the solution to. We then need to advance time by some Δt and end up with the set of differential equations defined by the Euler method as

$$H(x+\Delta x, t+\Delta t) \approx H(x,t) + \frac{\partial H}{\partial x}\Delta x + \frac{\partial H}{\partial t}\Delta t.$$

Note that we know that we have $H(x,t) = 0$ as we started with a solution! Additionally, we require that $H(x+\Delta x, t+\Delta t) = 0$ and we know what our time step Δt is. Thus, we need only solve the system

$$\frac{\partial H}{\partial x}\Delta x = -\frac{\partial H}{\partial t}\Delta t$$

for Δx. We then update our values of t and x by incrementin by Δt and Δx and then implement Newton to find values of t and x that lie on our curve. Finally, we step over all Δt until we get to $t = 1$ at which point x will be a solution to our original system!

1.1.1 Euler Step

We write a function to compute a Euler step.

[4]:
```
function euler_step(H::System, Δt::Float64, x::Vector, t::Float64)

    #make four element Vector
    xt = append!(x, t)
    #get indices for vector subsetting
    index = []
    for i in 2:length(x)
        append!(index, i)
    end

    #make jacobian and separate out dH/dx and dH/dt
    J = jacobian(H, xt)
```

```
    ∂H∂x = J[:,index]
    ∂H∂t = J[:,1]

    #set up system and solve for Δx
    Δx = ∂H∂x \ -(∂H∂t * Δt)

    #update values for x and t
    x = xt[index .- 1] + Δx
    t = t + Δt

    return x, t

end
```

[4]: `euler_step (generic function with 1 method)`

Notice that this function can take systems / vectors of any length and is thus a general solution. Let's just make sure this works in practice as follows

[5]:
```
time = 0.0;
Δt = 0.1;

x = [1.0, 2.0]

@var x1 x2 t
f1 = t * x1^2 + x2
f2 = x1 + x2 * t

F = System([f1, f2])

start, time = euler_step(F, Δt, x, time)
start
```

[5]:
```
2-element Vector{Float64}:
 1.1333333333333333
 1.8666666666666667
```

and we see that this works to first order approximation.

1.1.2 Newton Method

We write a function to implement the newton method. This implementation requires `HomotopyContinuation.jl`.

[6]:
```
function newton_solve(start::Vector, F::System, tolerance::Float64 = 1e-15)

    #evaluate initial point
    tol = evaluate(F, start)
```

```
    #check if initial point is solution
    not_minimized = true
    if tol == zeros(length(tol))
        not_minimized = false
    end

    i = 0
    while (not_minimized && i < 501)
        #test for convergence
        if i == 500
            error("Algorithm did not converge!")
        end

        #solve linear system
        J = jacobian(F, start)
        start_new = J \ -evaluate(F, start)

        #update values
        for i in 1:length(start)
            start[i] = start_new[i] + start[i]
        end
        tol = evaluate(F, start)

        #test if tolerance is reached
        not_minimized = false
        for i in 1:length(tol)
            if abs(tol[i]) > tolerance
                not_minimized = true
            end
        end
        i += 1
    end

    return start
end
```

[6]: `newton_solve (generic function with 2 methods)`

We note that this function also is general and can solve any problem. We verify this works on the complex numbers by

[7]:
```
@var x y
f1 = x^2 + y^3 - 1
f2 = 2*x - y
F = System([f1, f2])
p = randn(ComplexF64, 2)

newton_solve(p, F)
```

```
[7]: 2-element Vector{ComplexF64}:
      -0.2933069324711349 - 0.4298373375930675im
      -0.5866138649422697 - 0.859674675186135im
```

and check our answer using `HomotopyContinuation.jl`

```
[8]: S = System([f1, f2])
     result = solve(S)
```

```
Tracking 3 paths... 100%|| Time:
0:00:17
  # paths tracked:                  3
  # non-singular solutions (real):  3 (1)
  # singular endpoints (real):      0 (0)
  # total solutions (real):         3 (1)
```

```
[8]: Result with 3 solutions
     ======================
     • 3 paths tracked
     • 3 non-singular solutions (1 real)
     • random_seed: 0x5093c109
     • start_system: :polyhedral
```

```
[9]: solutions(result)
```

```
[9]: 3-element Vector{Vector{ComplexF64}}:
      [-0.2933069324711348 - 0.4298373375930676im, -0.5866138649422696 -
     0.8596746751861353im]
      [0.4616138649422696 - 2.407412430484045e-35im, 0.9232277298845392 -
     4.81482486096809e-35im]
      [-0.2933069324711348 + 0.4298373375930676im, -0.5866138649422696 +
     0.8596746751861353im]
```

1.1.3 Path Tracking and Solving!

We finally combine the two functions and return to the original problem to solve the system of two equations. We will use

$$f(x,y) = \begin{pmatrix} f_1(x,y) \\ f_2(x,y) \end{pmatrix}$$

Notice that we need some $g(x)$. Let's use

$$g(x,y) = \begin{pmatrix} x^6 - 1 \\ y^3 - 1 \end{pmatrix}$$

which we *know* has solutions

$$x = e^{ik\frac{2\pi}{6}}, k = 0, 1, 2, 3, 4, 5$$

and

$$y = e^{i\ell\frac{2\pi}{3}}, \ell = 0, 1, 2$$

and we see that we will have 18 total solutions. Let's implement this model and find the solution when $k = 0, \ell = 0$ to see if we get a solution that `HomotopyContinuation.jl` got.

[10]:
```
function path_track(H::System, start::Vector, Δt::Float64)

    time = 0.0
    tot_steps = 1 / Δt

    i = 0
    while i < tot_steps
        start, time = euler_step(H, Δt, start, time)
        start = newton_solve(start, System(evaluate(H, [time, x, y])), 1e-12)
        i += 1
    end

    return start;
end
```

[10]:
```
path_track (generic function with 1 method)
```

Notice that this function is *not* general and will only work for two-variable systems in variables x and y. This can easily be modified by small changes to our code. We then evaluate our function

[11]:
```
#construct System
@var x y t
fg1 = ((x^4 + y^4 - 1) * (x^2 + y^2 - 2) + x^5 * y) * t + (1 - t) * (x^6 - 1)
fg2 = (x^2 + 2*x*y^2 - 2*y^2 - 1/2) * t + (1 - t) * (y^3 - 1)
FS = System([fg1, fg2])

start_sol = [exp(im*0*π/6), exp(im*0*π/3)]

Δt = 0.001

path_track(FS, start_sol, Δt)
```

[11]:
```
2-element Vector{ComplexF64}:
  0.8209788924342625 - 1.2096476015712636e-18im
 -0.6971326459489465 - 3.947676315116146e-17im
```

Note that this is one of the points that `HomotopyContinuation` got! Let's now find *all* of the points by defining yet another function that scans all the first points we give it and then solves.

[12]:
```
function homotopy_solve(H::System)

    #set time step
    Δt = 0.001

    #open array for appending solutions
    solutions = []

    #compute all solutions
    #this only works for two equations which have degrees of 6 and 3␣
 ↪respectively
    for i in 0:5
        for j in 0:2
            start = [exp(i*im*2*π/6), exp(j*im*2*π/3)]
            sol = path_track(H, start, Δt)
            push!(solutions, sol)
        end
    end

    return solutions

end
```

[12]:
```
homotopy_solve (generic function with 1 method)
```

Notice that, again, our function is *not* general. Now it can only solve the two variable case with polynomial degrees of six and three as our problem is. This can be easily changed by small modifications of our code. We then find the solutions by implementing our method.

[13]:
```
#solve them thangs
sol2 = homotopy_solve(FS)
```

[13]:
```
18-element Vector{Any}:
 ComplexF64[0.8209788924342625 - 1.2096476015712636e-18im, -0.6971326459489465 -
3.947676315116146e-17im]
 ComplexF64[0.9443571312488817 + 0.3118635017972665im, 0.3208381852903997 +
0.9677296009728291im]
 ComplexF64[0.9443571312488817 - 0.3118635017972665im, 0.3208381852903997 -
0.9677296009728291im]
 ComplexF64[1.713294137558267 + 0.5813863945698415im, 0.04751413478085409 -
1.2527929510076878im]
 ComplexF64[0.07565391048031049 + 0.9487419814734106im, -0.24800445792173492 +
0.6838307098593375im]
 ComplexF64[0.8999179208471724 + 1.577903428960801e-20im, -1.244182761342273 -
2.0837887478012815e-18im]
 ComplexF64[0.06944255588971956 + 1.0734210145259258im, 0.2852544211796043 -
0.7076856100161801im]
 ComplexF64[-1.0665536440076715 + 0.1424944286215246im, -0.3998429331325369 +
```

```
0.07871238147493335im]
 ComplexF64[1.0866676911062139 + 0.32907006699787644im, -0.2404810670866118 -
1.1350215823993675im]
 ComplexF64[0.8209788924342625 - 1.2096476015712636e-18im, -0.6971326459489465 -
3.947676315116146e-17im]
 ComplexF64[-1.6714213928380037 - 1.0186441737974984e-16im, 0.655205185872041 -
1.3262203461352321e-17im]
 ComplexF64[-1.6714213928380037 + 1.0186441737975248e-16im, 0.655205185872041 +
1.3262203461352648e-17im]
 ComplexF64[0.06944255588971956 - 1.0734210145259258im, 0.2852544211796043 +
0.7076856100161801im]
 ComplexF64[1.0866676911062139 - 0.32907006699787644im, -0.2404810670866118 +
1.1350215823993675im]
 ComplexF64[-1.0665536440076715 - 0.1424944286215246im, -0.3998429331325369 -
0.07871238147493335im]
 ComplexF64[1.713294137558267 - 0.5813863945698415im, 0.04751413478085409 +
1.2527929510076878im]
 ComplexF64[0.8999179208471724 - 1.577903428960801e-20im, -1.244182761342273 +
2.0837887478012815e-18im]
 ComplexF64[0.07565391048031049 - 0.9487419814734106im, -0.24800445792173492 -
0.6838307098593375im]
```

Notice that these are the same solutions that `HomotopyContinuation` got! Discrepancies arrise for values that are effectively zero, otherwise our answer is exactly the same as the above solution.

2 Conclusions

We were able to implement concepts of Homotopy Continuation to find all of the solutions to our system of equations. Of course, making our functions general, i.e. able to solve *any* system of equations is possible, but putting the work in to implement this is rather pointless as `HomotopyContinuation.jl` does a fantastic job of solving any system of equations in just a few lines of code (and we're already using functions from `HomotopyContinuation.jl` for our implementation). The above implementation shows, however, how powerful techniques in Homotopy Continuation can be in solving polynomial equations and how easy these methods are to implement.