UNIVERSITÀ DEGLI STUDI DI PISA

Dipartimento di Matematica

Corso di Laurea Triennale in Matematica

Laboratorio Computazionale

Parallel Homotopy Continuation in Julia

Studente: Francesco Minnocci
Matricola: 600455

ANNO ACCADEMICO 2022 - 2023

Contents

1 Introduction

Homotopy Continuation is a numerical method for solving systems of polynomial equations. It is based on the idea of "deforming" a given system of equations into a simpler one, whose solutions are known, and then tracking the solutions of the original system as the deformation is undone.

In this project, the method will be implemented in the Julia programming language, making use of parallel computing in order to speed multiple root finding. The method is described in detail in [1], which was the primary source for this report.

2 Homotopy Continuation

We will only consider *square* systems of polynomial equations, i.e. systems of n polynomial equations in n variables, although or over- or under-determined systems can often be solved by reducing them to square systems, by respectively choosing a suitable square subsystem or adding equations. Morever, we will restrict ourselves to systems with isolated solutions, i.e. zero-dimensional varieties.

There are many ways to choose the "simpler" system, from now on called a *start system*, but in general we can observe that, by Bezout's theorem, a system $F = (f_1, \dots, f_n)$ has at most $D := d_1 \dots d_n$ solutions, where d_i is the degre of $f_i(x_1, \dots, x_n)$. So, we could build a start system of the same size and whose polynomials have the same degrees, but whose solutions are easy to find, and thus can be used as starting points for the method.

For instance, the system $G = (g_1, \dots g_n)$, where

$$g_i(x_1, \dots x_n) = x_i^{d_i} - 1,$$

is such a system, since it has exactly the D solutions

$$\left\{ \left(e^{\frac{k_1}{d_1} 2\pi i}, \dots, e^{\frac{k_n}{d_n} 2\pi i} \right), \text{ for } 0 \leq k_i \leq d_i - 1 \text{ and } i = 1, \dots, n \right\}.$$

2.1 Choosing the homotopy

The deformation between the original system and the start system is a *homotopy*, for instance the convex combination of F and G

$$H(x, t) = (1 - t)F(x) + tG(x), \tag{1}$$

where $x := (x_1, \dots, x_n)$ and $t \in [0, 1]$. This is such that the roots of $H(x, 0) = G(x)$ are known, and the roots of $H(x, 1) = F(x)$ are the solutions of the original system. Therefore, we can implicitly define a curve $z(t)$ in $\mathbb{C}^n$ by the equation

$$H(z(t), t) = 0, \tag{2}$$

so that in order to approximate the roots of F it is enough to numerically track $z(t)$.

To do so, we derive the expression (2) with respect to t, and get the *Davidenko Differential Equation*

$$\frac{\partial H}{\partial z} \frac{\mathrm{d}z}{\mathrm{d}t} + \frac{\partial H}{\partial t} = 0,$$

where $\frac{\partial H}{\partial z}$ is the Jacobian matrix of H with respect to z:

$$\frac{\partial H}{\partial z} = \begin{pmatrix} \frac{\partial H_1}{\partial z_1} & \cdots & \frac{\partial H_1}{\partial z_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial H_n}{\partial z_1} & \cdots & \frac{\partial H_n}{\partial z_n} \end{pmatrix}.$$

This can be rewritten as

$$\dot{z} = -\frac{\partial H}{\partial z}^{-1} \frac{\partial H}{\partial t}. \tag{3}$$

This is a system of n first-order differential equations, which can be solved numerically for $z(t)$ as an initial value problem, and is called *path tracking*.

2.1.1 Gamma trick

While (1) is a fine choice of a homotopy, it's not what it's called a *good homotopy*: in order to ensure that the solution paths $z(t)$ for different roots

- have no singularities, i.e. never cross each other for $t > 0$ (at $t = 0$, F could have singular solutions), and
- don't go to infinity for $t \to 0$ (F could have a solution at infinity),

we can employ the *Gamma trick*: this consists in modifying the linear homotopy (1) by susbtituting the parameter $t \in [0, 1]$ with a complex curve $q(t)$ connecting 0 and 1:

$$q(t) = \frac{\gamma t}{\gamma t + (1 - t)},$$

where $\gamma \in (0, 1)$ is a random complex parameter.This "probability one" procedure, i.e. for any particular system choosing γ outside of a finite amount of lines through the origin ensures that we get a good homotopy, basically because of the finiteness of the branch locus of the homotopy. After substituting, we get

$$H(x, t) = \frac{(1 - t)}{\gamma t + (1 - t)} F(x) + \frac{\gamma t}{\gamma t + (1 - t)} G(x),$$

and clearing denominators, here's our final homotopy:

$$H(x, t) = (1 - t)F(x) + \gamma t G(x). \tag{4}$$

2.2 Tracking down the roots

We now want to track down individual roots, following the solution paths from a root z_0 of the start system by solving the initial value problem associated to the Davidenko differential equation (3) with starting value z_0 and t ranging from 1 to 0.

This will be done numerically, using a first-order predictor-corrector tracking method, which consists in first using Euler's method to get an approximation $\widetilde{z}_i$, and then using Newton's method to correct it using equation (2) so that it becomes a good approximation z_i of the next value of the solution path.

2.2.1 Predictor: Euler's method

Recall that Euler's method consists in approximating the solution of the initial value problem associated to a first-order ordinary differential equations

$$\begin{cases} \dot{z} = f(z,t) \\ z(t_0) = z_0 \end{cases}$$

by the sequence of points $(z_i)_{i\in\mathbb{N}}$ defined by the recurrence relation

$$z_{i+1} = z_i + h \cdot f(z_i, t_i),$$

where h is the step size. In our case, we have

$$f(z,t) = -\left(\frac{\partial H}{\partial z}(z,t)\right)^{-1} \frac{\partial H}{\partial t}(z,t)$$

and $t_0 = 1$, since we track from 1 to 0. For the same reason, we set

$$t_{i+1} = t_i - h.$$

2.2.2 Corrector: Newton's method

Since we want to solve

$$H(z,t) = 0,$$

we can use Newton's method to improve the approximation $\widetilde{z}_i$ obtained by Euler's method to a solution of such equation. This is done by moving towards the root of the tangent line of H at the current approximation, or in other words through the iteration

$$z_{i+1} = z_i - \left(\frac{\partial H}{\partial z}(z_i, t_{i+1})\right)^{-1} H(z_i, t_{i+1}),$$

where this time $z_0 = \widetilde{z}_i$, with $\widetilde{z}_i$ and t_{i+1} obtained from the i-th Euler step.

Usually, only a few steps of Newton's method are needed; we will use a fixed number of 5 iterations. At this point, we use the final value of the Newton iteration as the starting value for the next Euler step.

2.2.3 Adaptive step size

In order to improve the efficiency of the method, we will use an adaptive step size, which will be based on the norm of the residual of the Newton iteration. If the desired accuracy is not reached, for instance when the norm of $H(z_i, t_i)$ is bigger than 10^{-8}, then we halve the step size; if instead we have 5 "successful" iterations in a row, we double the step size, as implemented in Appendix A.

3 Parallelization

3.1 Multithreading

When testing the method, we tried to use multithreading to speed up the computation. This was done in Julia by using the `Threads.@threads` macro, which automatically distributes the

work of a `for` loop among the available threads. However, in the case of looping over multiple roots, this didn't improve the performance, as the overhead of the multithreading was too big compared to the actual computation time, as the systems were too small to benefit from this kind of parallelization, as can be seen by the results in Appendix B.

3.2 MPI

Next, we tried to use MPI to parallelize the tracking of the roots. This was done by using the `MPI.jl` [2] package, which provides a Julia interface to the MPI library.

4 Appendix A: Implementation

4.1 Julia code

Listing 1: solve.jl

```julia
# External dependencies
using TypedPolynomials
using LinearAlgebra

# Local dependencies
include("random_poly.jl")
include("start-system.jl")
include("homotopy.jl")
include("euler-newton.jl")
include("adapt-step.jl")
include("plot.jl")
using .RandomPoly
using .StartSystem
using .Homotopy
using .EulerNewton
using .AdaptStep
using .Plot

# Main homotopy continuation loop
function solve(F, (G, roots) = start_system(F), maxsteps = 1000)
  H=homotopy(F,G)
  solutions = []
  step_array = []

  Threads.@threads for r in roots
  #  for r in roots
    println("New root")
    t = 1.0
    step_size = 0.01
    x0 = r
    m = 0
    steps = 0

    while t > 0 && steps < maxsteps
      x0 = en_step(H, x0, t, step_size)
      (m, step_size) = adapt_step(H, x0, t, step_size, m)
      t -= step_size
      steps += 1
    end
    push!(solutions, x0)
    push!(step_array, steps)
  end

  return (solutions, step_array)
end

# Input polynomial systems
#  @polyvar x y
#  C = [x^3 - y + 5x^2 - 10, 2x^2 - y - 10]
```

```
#  Q = [x^2 + 2y, y - 3x^3]
#  F = [x*y - 1, x^2 + y^2 - 4]
#  T = [x*y - 1, x^2 + y^2 - 2]
dimension = 2
R = random_system(2, 2)
println(R)

#  (sC, stepsC) = solve(C)
#  (sQ, stepsQ) = solve(Q)
#  (sF, stepsF) = solve(F)
#  (sT, stepsT) = solve(T)
(sR, stepsR) = solve(R)

#  println("C: ", stepsC)
#  println("Q: ", stepsQ)
#  println("F: ", stepsF)
#  println("T: ", stepsT)
println("R: ", stepsR)

#  sC = filter(u -> imag(u[1]) < 0.1 && imag(u[2]) < 0.1, sC)
#  sQ = filter(u -> imag(u[1]) < 0.1 && imag(u[2]) < 0.1, sQ)
#  sF = filter(u -> imag(u[1]) < 0.1 && imag(u[2]) < 0.1, sF)
#  sT = filter(u -> imag(u[1]) < 0.1 && imag(u[2]) < 0.1, sT)
sR = filter(u -> imag(u[1]) < 0.1 && imag(u[2]) < 0.1, sR)

vars = variables(R)
println("solutions: ", sR)
println([LinearAlgebra.norm([f(vars=>s) for f in R]) for s in sR])

# Plotting the system and the real solutions
ENV["GKSwstype"]="nul"
#  plot_real(sC, C, 6, 12, "1")
#  plot_real(sQ, Q, 2, 2, "2")
#  plot_real(sF, F, 4, 4, "3")
#  plot_real(sT, T, 4, 4, "4")
plot_real(sR, R, 5, 5, "random")
```

Listing 2: start-system.jl

```
module StartSystem
  using TypedPolynomials

  export start_system

  # Define start system based on total degree
  function start_system(F)
    degrees = [maxdegree(p) for p in F]
    G = [x_i^d - 1 for (d, x_i) in zip(degrees, variables(F))]
    r = [[exp(2im*pi/d)^k for k=0:d-1] for d in degrees]
    roots = vec([collect(root) for root in collect(Iterators.product(r...))])
    return (G, roots)
  end
end
```

Listing 3: homotopy.jl

```
module Homotopy
  export homotopy

  # Define a straight-line homotopy between the two systems
  function homotopy(F, G)
    γ = cis(2π * rand())
    function H(t)
      return [(1 - t) * f + γ * t * g for (f, g) in zip(F, G)]
    end
    return H
```

```
  end
end
```

Listing 4: homogenize.jl

```
module Homogenize
  using TypedPolynomials

  export homogenize, homogenized_start_system

  function homogenize(F)
    @polyvar h
    return  [sum([h^(maxdegree(p)-maxdegree(t))*t for t in p.terms]) for p in F
     ]
  end

  function homogenized_start_system(F)
    degrees = [maxdegree(p) for p in F]
    @polyvar h
    G = [x_i^d - h^d for (d, x_i) in zip(degrees, variables(F))]
    r = [[exp(2im*pi/d)^k for k=0:d-1] for d in degrees]
    roots = vec([vcat(collect(root), 1) for root in collect(Iterators.product(r
     ...))])
    return (G, roots)
  end
end
```

Listing 5: euler-newton.jl

```
module EulerNewton
  using LinearAlgebra
  using TypedPolynomials

  export en_step

  # Euler-Newton predictor-corrector
  function en_step(H, x, t, step_size)

    # Predictor step
    vars = variables(H(t))
    # Jacobian of H evaluated at (x,t)
    JH = [jh(vars=>x) for jh in differentiate(H(t), vars)]
    # ∂H/∂t is the same as γG-F=H(1)-H(0) for our choice of homotopy
    Δx = JH \ -[gg(vars=>x) for gg in H(1)-H(0)]
    xh = x + Δx * step_size

    # Corrector step
    JHh=differentiate(H(t-step_size), vars)
    for _ in 1:5
      JH = [jh(vars=>xh) for jh in JHh]
      Δx = JH \ -[h(vars=>xh) for h in H(t-step_size)]
      xh = xh + Δx
    end

    return xh
  end
end
```

Listing 6: adapt-step.jl

```
module AdaptStep
  using LinearAlgebra
  using TypedPolynomials
```

```
  export adapt_step

  # Adaptive step size
  function adapt_step(H, x, t, step, m)
    Δ = LinearAlgebra.norm([h(variables(H(t))=>x) for h in H(t-step)])
    if Δ > 1e-8
      step = 0.5 * step
      m = 0
    else
      m+=1
      if (m == 5) && (step < 0.05)
        step = 2 * step
        m = 0
      end
    end

    return (m, step)
  end
end
```

Listing 7: plot.jl

```
module Plot
  using Plots, TypedPolynomials

  export plot_real

  function plot_real(solutions, F, h, v, name)
    plot(xlim = (-h, h), ylim = (-v, v), aspect_ratio = :equal)
    contour!(-h:0.1:h, -v:0.1:v, (x,y)->F[1](variables(F)=>[x,y]), levels=[0],
      cbar=false, color=:cyan)
    contour!(-h:0.1:h, -v:0.1:v, (x,y)->F[2](variables(F)=>[x,y]), levels=[0],
      cbar=false, color=:green)
    scatter!([real(sol[1]) for sol in solutions], [real(sol[2]) for sol in
      solutions], color = "red", label = "Real solutions")

    png(joinpath("plots", "solutions" * name))
  end
end
```

4.2 Hardware

For the single-threaded runs, the code was run on a laptop with an Intel Core i7-3520M CPU @ 3.60GHz and 6 GB of RAM.

For the multithreaded runs, it was run on a desktop with an AMD FX-8350 CPU @ 4.00GHz with 4 cores and 8 threads, and 12 GB of RAM.

For the parallel runs, it was run on a cluster with N nodes, each with X CPUs @ Y.ZGHz with M cores and P threads each, and Q GB of RAM.

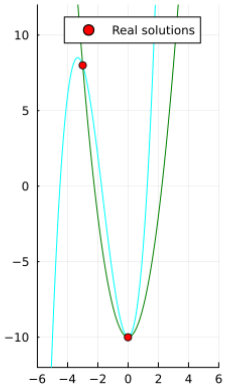
5 Appendix B: Results

5.1 Multithreading

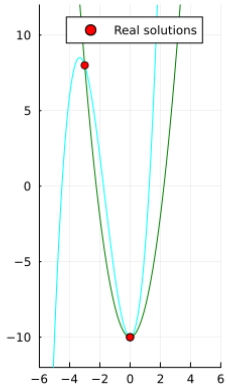
Here are the plots for the solutions of four different 2x2 systems, with the single-threaded version next to the multithreaded one:

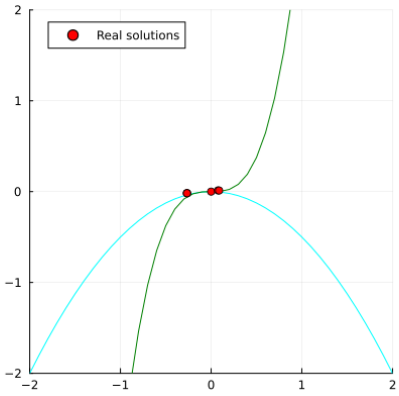
Single-threaded

Multithreaded

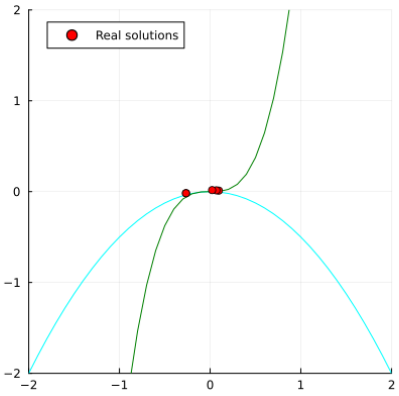


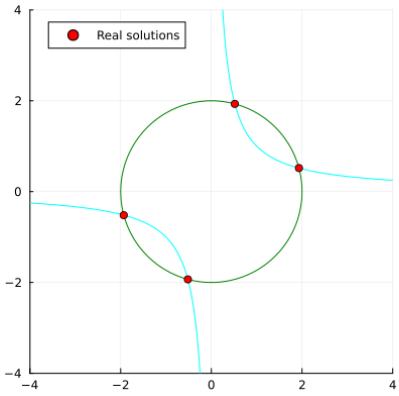
$$\begin{cases} x^3 + 5x^2 - y - 1 \\ 2x^2 - y - 1 \end{cases}$$



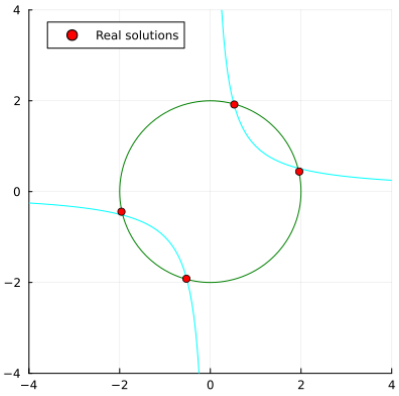


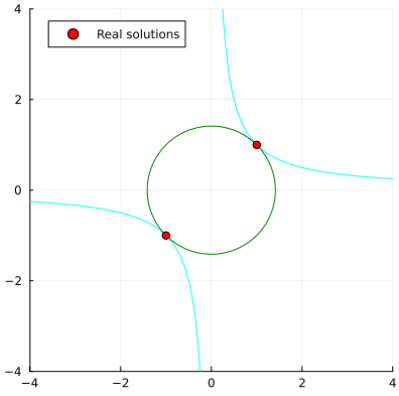
$$\begin{cases} x^2 + 2y \\ y - 3x^3 \end{cases}$$



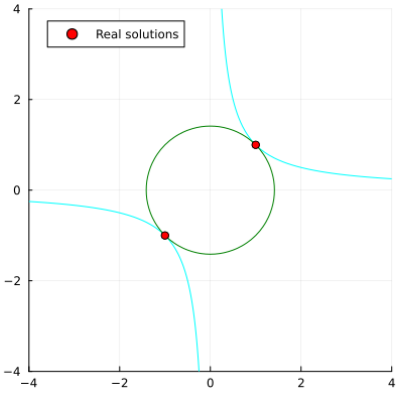


$$\begin{cases} x^2 + y^2 - 4 \\ xy - 1 \end{cases}$$





$$\begin{cases} x^2 + y^2 - 2 \\ xy - 1 \end{cases}$$



References

[1] Bates, Daniel J. *Numerically solving polynomial systems with Bertini*. SIAM, Society for Industrial Applied Mathematics, 2013.

[2] Simon Byrne, Lucas C. Wilcox, and Valentin Churavy (2021) "MPI.jl: Julia bindings for the Message Passing Interface". JuliaCon Proceedings, 1(1), 68, doi: 10.21105/jcon.00068