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, which is particularly suited for scientific computing. The primary source for this report is [1], where the method is explained in much more detail.

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 squaring it by adding equations. Morever, we will restrict ourselves to systems which have 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 degree of $f_i(x_1, \dots, x_n)$.

Therefore, we can 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 its zero locus is obtained by combining the d_i-th roots of unity in each variable, which are exactly D points:

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

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 (the reason why we place the start system at $t = 0$ and the original system at $t = 1$ is that we need higher numerical precision for the solutions of the original system, and there are more floating point numbers near to $t = 0$; see [1], p. 33). 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, which 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 $t \to 0$ (as 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, such as

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

where $\gamma \in (0, 1)$ is a random complex parameter.

This is a "probability one" procedure, i.e. for any particular system we can choose γ outside of a finite amount of rays through the origin to ensure that we get a good homotopy, basically because of the finiteness of the branch locus of the homotopy. After substituting, we have

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

and by clearing denominators, we get our final choice of homotopy:

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

2.2 Tracking down the roots

We then need 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, by using a first-order predictor-corrector tracking method, whose typical iteration goes like this:

- **Predictor:** we first apply Euler's method to get an approximation $\widetilde{z}_i$ of the next value of the solution path;
- **Corrector:** we then use Newton's method to correct $\widetilde{z}_i$ using equation (2), so that it becomes a good approximation z_i of the next value of the solution path.

In the following sections, we go into more detail on each of these steps.

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 system of 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 the case of the Davidenko equation (3), 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 are tracking 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. 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 chose a fixed number of 5 iterations. At which 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 is based on the norm of the residual of Newton's iteration. If the desired accuracy is not reached (say, 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.

3 Testing the method

To test the method's scalability, we first launched it on a single-threaded machine, then one a multi-threaded one, and finally parallelized it on a Cluster.

The latter was done by using the Julia package *Distributed.jl* to parallelize the tracking of the roots on separate nodes, and the `SlurmClusterManager` package, which allows to run Julia code using the `Slurm` workload manager.

In order to scale the method to larger systems, we also implemented a random polynomial generator which can be found in random-poly.jl; this was used to evaluate the performance of the parallel implementation, by generating square systems of polynomials with normally distributed coefficients, each polynomial having total degree less or equal to a fixed maximum degree.

The single-threaded machine and multi-threaded tests (which used the `@threads` macro from the *Threads.jl* package on the root tracking `for` loop in the file solve.jl) were run in order to visualize the real solutions of small (2x2) systems: here, multi-threaded runs didn't improve the performance on these smaller systems, as the overhead of multi-threading was too big compared to the actual computation time.

However, when testing a parallel implementation on larger randomly generated systems we observed an improvement in execution times on larger systems compared to the single-node runs, as we show in the Results section.

The Julia implementation for the tests described above can be found in Appendix B, while the hardware specifications are listed in Appendix A.

4 Possible Improvements

4.1 Homogenized Coordinates

Since our start systems have the maximum number of solutions for its degree, some of them might converge to a point at infinity of our original system. In our current implementation, we waste time by tracking them until reaching the maximum number of iterations.

To better treat such cases, we could view the system inside an affine patch of the projective plane, and using homogenized coordinates detect when a solution is going to infinity. This would involve homogenizing both systems and modifying the path-tracking algorithm for the detection of a point going to infinity.

4.2 Predictor-Corrector

Our (un)specific choice of predictor could be unsuitable for badly-conditioned systems; other software implementations of the homotopy continuation method use more accurate and numerically stable predictors, such as Runge-Kutta methods [3].

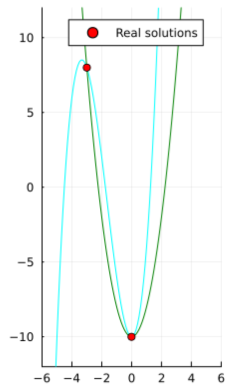
5 Appendix A: Results

5.1 Single- and Multi-threaded

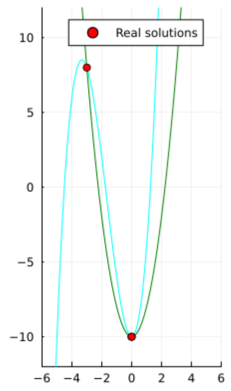
Below are the plots of four different 2x2 systems for the single- (laptop) and multi- (desktop) threaded runs, with the real solutions being shown in red:

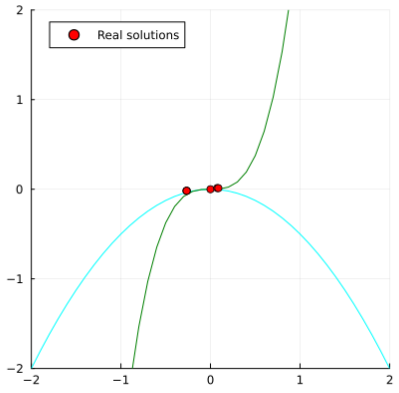
Single-threaded

Multithreaded

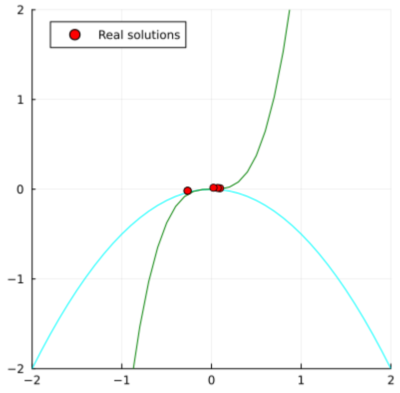


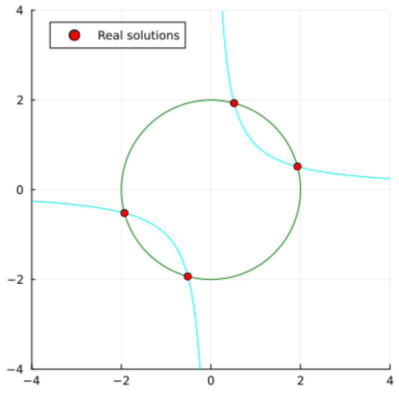
$$\begin{cases} x^3 + 5x^2 - y - 1 \\ 2x^2 - y - 1 \end{cases}$$



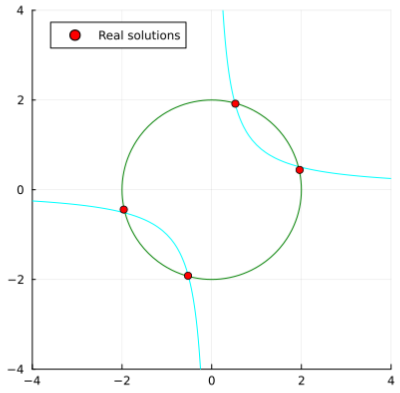


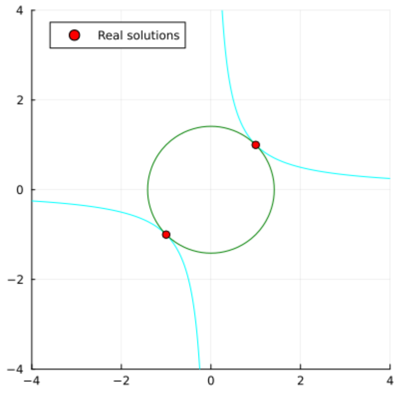
$$\begin{cases} x^2 + 2y \\ y - 3x^3 \end{cases}$$



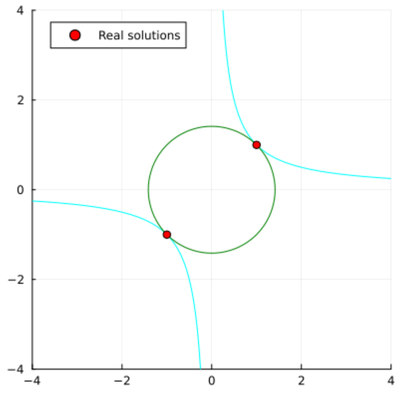


$$\begin{cases} x^2 + y^2 - 4 \\ xy - 1 \end{cases}$$





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



5.2 Parallelization

The following figure compares the execution times of the `solve` function in solve.jl on the cluster, on a single node and on 20 nodes (using 1 or 2 threads per node).

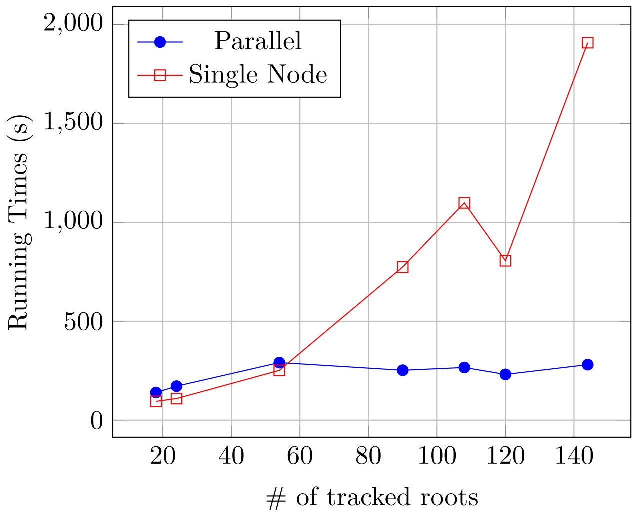


Figure 1: Performance comparison of parallel path tracking on a cluster.

As we can see from the plot, the parallel implementation appears to scale well with the number of tracked roots, and is faster than the single-node implementation for larger systems.

6 Appendix B: Implementation

6.1 Julia code

Listing 1: solve.jl

```julia
# External deps
using LinearAlgebra
using TypedPolynomials
using Distributed
#  using SlurmClusterManager
#  slurm_manager = SlurmManager()
#  addprocs(slurm_manager)

# Local deps
include("random-poly.jl")
include("plot.jl")
using .RandomPoly
using .Plot
@everywhere begin
  include("start-system.jl")
  include("homotopy.jl")
  include("euler-newton.jl")
  include("adapt-step.jl")
end
# Macros defined in an @everywhere block aren't available inside it
@everywhere begin
  using .StartSystem
  using .Homotopy
  using .EulerNewton
  using .AdaptStep
end

#  ciao sto facendo delle modifiche al codice

@everywhere function compute_root(H, r, maxsteps=200)
  t = 1.0
  step_size = 0.001
  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
  return (x0, steps)
end

# Main homotopy continuation loop
function solve(F, G, roots)
  H = homotopy(F, G)

  result = Array{Future}(undef, length(roots))
  for i in eachindex(roots)
    result[i] = @spawnat :any compute_root(H, roots[i])
  end

  sols = Array{ComplexF64,2}(undef, length(roots), length(F))
  steps = Array{Int64}(undef, length(roots))
  for i in eachindex(roots)
    (solution, step_array) = fetch(result[i])
    sols[i, :] = solution
    steps[i] = step_array
  end

  return (sols, steps)
end

#  @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]

R = random_system(6, 6)
println("System: ", R)
(G, roots)=start_system(R)
println("Number of roots: ", length(roots))

# Parallel execution
println("PARALLEL")
@time begin
  (sol, steps) = solve(R, G, roots)
end
println("Number of steps: ", steps)
# converting sR to array of arrays instead of a matrix
sol = [sol[i, :] for i in 1:length(sol[:, 1])]
sol = filter(u -> imag(u[1]) < 0.1 && imag(u[2]) < 0.1, sol)
sol = map(u -> real.(u), sol)
vars = variables(R)
println("Solutions: ", sol)
println("Norms (lower = better): ", [norm([f(vars => s) for f in R]) for s in
     sol])

#  # Single execution
#  println("SINGLE")
#  wait(rmprocs(workers()))
#  @time begin
  #  (sol, steps) = solve(R, G, roots)
#  end
#  println("Number of steps: ", steps)
#  # converting sR to array of arrays instead of a matrix
#  sol = [sol[i, :] for i in 1:length(sol[:, 1])]
#  sol = filter(u -> imag(u[1]) < 0.1 && imag(u[2]) < 0.1, sol)
#  sol = map(u -> real.(u), sol)
#  vars = variables(R)
#  println("Solutions: ", sol)
#  println("Norms (lower = better): ", [norm([f(vars => s) for f in R]) for s
     in sol])

#  # See https://github.com/kleinhenz/SlurmClusterManager.jl/issues/11
#  finalize(slurm_manager)

# 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(sol, 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(1))
    # Jacobian of H evaluated at (x,t)
    JH = [jh(vars=>x) for jh in differentiate(H(t), vars)]
    # ∂H/∂t = γG-F = H(1)-H(0) for our homotopy; it doesn't depend on t
    δH_δt = [dh(vars=>x) for dh in H(1)-H(0)]
    Δx = JH \ -δH_δt
    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 = norm([h(variables(H(t))=>x) for h in H(t-step)])
    if Δ > 1e-10
      step = 0.5 * step
      m = 0
    else
      m+=1
      if (m == 4)
        step = 2 * step
        m = 0
      end
    end

    return (m, step)
  end
end
```

Listing 7: random-poly.jl

```
module RandomPoly
  export random_system

  using TypedPolynomials
  using Random
  using Distributions

  # Random polynomial of degree n in m variables
  function random_poly(n, m)
    x = [TypedPolynomials.Variable{Symbol("x[$i]")}() for i in 1:m]

    monomial_powers=collect(Iterators.product([0:n for _ in 1:m]...))
    monomials = [prod(x.^i) for i in monomial_powers if sum(i) <= n && sum(i) != 0]

    return sum(map(m -> rand(Normal()) * m, monomials))
  end

  # Generate a system of m random polynomials in m variables
  # of degree d_i randomly chosen between 1 and max_degree
  function random_system(m, max_degree)
    d = rand(1:max_degree, m)
    random_polys = [random_poly(d[i], m) for i in 1:m]

    return random_polys
  end
end
```

Listing 8: 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
```

6.2 Hardware

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

The multithreaded runs were tested on a desktop with an AMD FX-8350 CPU @ 4.00GHz with 4 cores and 8 threads, and 12 GB of RAM.

Finally, the parallel computations were run on a cluster with 20 nodes, each having a CPU @ 1.008GHz with 4 Performance cores, 2 efficiency cores and 4 GB of RAM.

References

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

[2] https://docs.julialang.org/en/v1/stdlib/Distributed

[3] Breiding, P., Timme, S. (2018). HomotopyContinuation.jl: A Package for Homotopy Continuation in Julia. In: Davenport, J., Kauers, M., Labahn, G., Urban, J. (eds) Mathematical Software – ICMS 2018. Lecture Notes in Computer Science, vol 10931. Springer, Cham.