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.

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 can use as a start system $G = (g_1, \dots g_n)$, where

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

Indeed, this system has exactly D solutions

$$\left\{(z_1, \dots, z_n),\ z_i = e^{\frac{2\pi i k}{d_i}} \text{ for } k = 0, \dots, d_i \text{ and } i = 1, \dots, n\right\}.$$

2.1 Choosing the homotopy

The deformation between the original system and the start system is a *homotopy*, for instance one of the form

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

where $x := (x_1, \dots, x_n)$. 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.

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

- 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*:

2.2 Tracking down the roots

2.2.1 Davidenko differential equation

2.2.2 Predictor: Euler's method

2.2.3 Corrector: Newton's method

3 Parallelization

4 Implementation

4.1 Julia code

Listing 1: solve.jl

```
# External dependencies
using TypedPolynomials

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

# Main homotopy continuation loop
function solve(F, (G, roots) = start_system(F), maxsteps=10000)
  #  F=homogenize(F)
  H=homotopy(F,G)
  solutions = []

  Threads.@threads for r in roots
    t = 1.0
    step_size = 0.01
    x0 = r
    m = 0
    steps = 0

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

  return solutions
end

# Input polynomial system
@polyvar x y
F = [x*y - 1, x^2 + y^2 - 4]
T = [x*y - 1, x^2 + y^2 - 2]
C = [x^3 - y + 5x^2 - 10, 2x^2 - y - 10]

sF = filter(u -> imag(u[1]) < 0.1 && imag(u[2]) < 0.1, solve(F))
sT = filter(u -> imag(u[1]) < 0.1 && imag(u[2]) < 0.1, solve(T))
```

```
sC = filter(u -> imag(u[1]) < 0.1 && imag(u[2]) < 0.1, solve(C))

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

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)]
    Δx = JH \ -[gg(vars=>x) for gg in H(1)-H(0)] # ∂H/∂t is the same as γG-F=H
     (1)-H(0) for our choice of homotopy
    xp = x .+ Δx * step_size

    # Corrector step
    for _ in 1:10
      JH = [jh(vars=>xp) for jh in differentiate(H(t+step_size), vars)]
      Δx = JH \ -[h(vars=>xp) for h in H(t+step_size)]
      xp = xp .+ Δx
      if LinearAlgebra.norm(Δx) < 1e-6
        break
      end
    end

    return xp
  end
end
```

Listing 6: adapt-step.jl

```
module AdaptStep
  using LinearAlgebra

  export adapt_step

  # Adaptive step size
  function adapt_step(x, x_old, step, m)
    Δ = LinearAlgebra.norm(x - x_old)
    if Δ > 0.1
      step = 0.5 * step
      m = 0
    else
      m+=1
      if (m == 5)
        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

5 Results

References

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