Introduction to Julia

1. the Language Julia
   - about Julia
   - a first session
2. Exploring Algorithms
   - defining functions
   - cobweb diagrams
3. Fractal Plots
   - the method of Laguerre
   - C, Python, and Julia

MCS 507 Lecture 9
Mathematical, Statistical and Scientific Software
Jan Verschelde, 11 September 2023

Introduction to Julia

Why Julia?

- Write efficient code with the ease of scripting.

  Scripting languages make programmers more productive,
  but at the expense of execution efficiency.

- A new language for mathematical computations.

  Although Python has the SciPy stack,
  its primary target is not computational science.

running Julia

Four ways to run Julia code:

1. online at `cocalc.com`
2. in a Jupyter notebook, with IJulia
3. an interactive terminal session, the Julia REPL
4. launch scripts (`.jl` extension) at the command prompt

about Julia

- Online documentation: `https://docs.julialang.org`
- J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah.
  Julia: A fresh approach to numerical computing.
  SIAM Review, 59(1):65–98, 2017.
- JuliaCon: `https://juliacon.org`
- Jose Storopoli, Rik Huijzer, Lazaro Alonso: *Julia Data Science*.
  First edition published 2021. `https://juliadatascience.io`
  Creative Commons Attribution-Noncommercial-ShareAlike 4.0 International
- Alan Edelman: *A programming language to heal the planet together: Julia*. TEDx Talks, TEDxMIT, 2020.
  `https://youtu.be/qGW0GT1rCvs`

Introduction to Julia

1. the Language Julia
   - about Julia
   - a first session
2. Exploring Algorithms
   - defining functions
   - cobweb diagrams
3. Fractal Plots
   - the method of Laguerre
   - C, Python, and Julia

compute all eigenvalues of a normal matrix

```
$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.9.1 (2023-06-07)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> using LinearAlgebra

julia> A = randn(100,100);

julia> L = eigvals(A)
100-element Vector{ComplexF64}:
 -10.695990252140913 + 0.0im
  -9.030738730904044 - 4.073862615555709im
  -9.030738730904044 + 4.073862615555709im
...
```

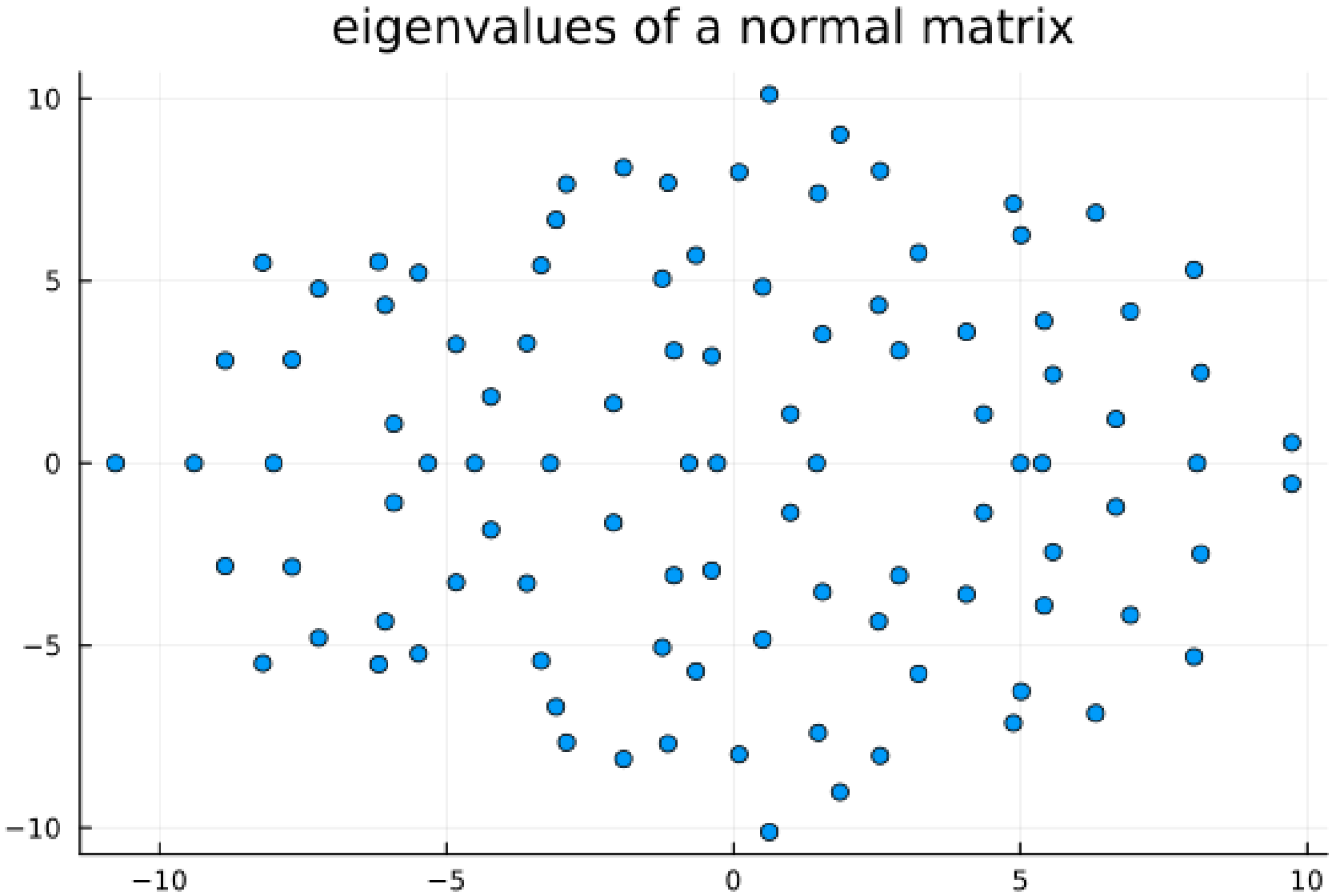
session continued in a Jupyter notebook

Let us look at a scatter plot of the eigenvalues.

```
using Plots

Lre = [real(x) for x in L]
Lim = [imag(x) for x in L]
scatter(Lre, Lim, label="",
        title="eigenvalues of a normal matrix")
```

the scatter plot



a first exercise

Exercise 1:

Instead of `randn(100, 100)`, do `rand(100, 100)` to make `A`.

Make a scatter plot of the eigenvalues of your random `A`.

Describe what you see.

Introduction to Julia

1. the Language Julia
   - about Julia
   - a first session
2. Exploring Algorithms
   - defining functions
   - cobweb diagrams
3. Fractal Plots
   - the method of Laguerre
   - C, Python, and Julia

defining functions

1. The documentation string comes *before* the definition.
2. The types of the arguments can be specified.
3. The code is written in a MATLAB like syntax.

the modified Gram-Schmidt method

Consider the modified Gram-Schmidt method to compute the QR decomposition of a matrix.

- With 64-bit floats:

```
function gsqr(A::Array{Float64,2})
```

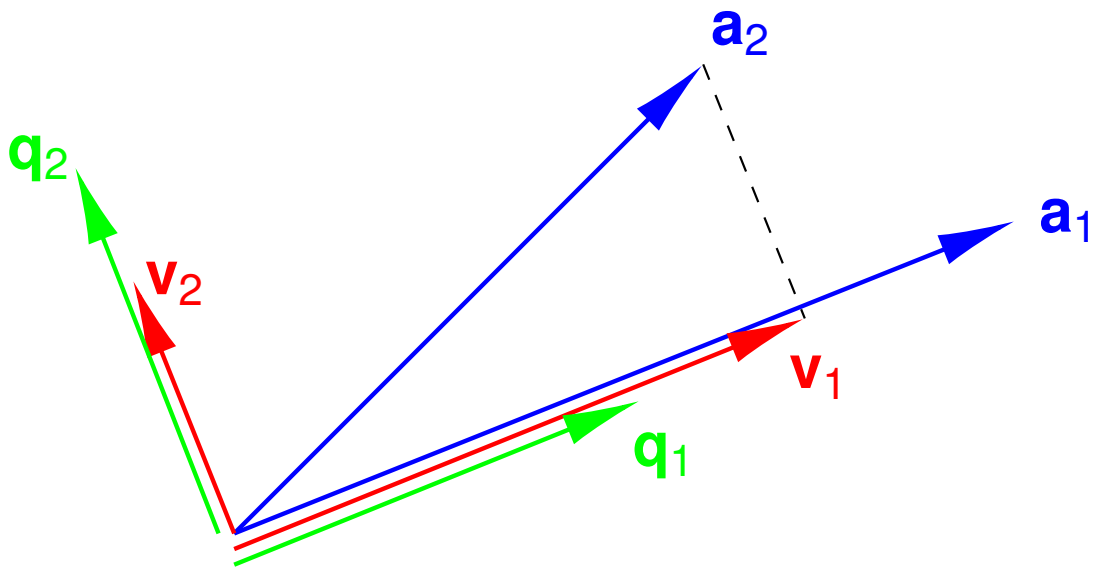
- With arbitrary precision floats:

```
function gsqr(A::Array{BigFloat,2})
```

The type `BigFloat` is part of the standard library.

a geometric interpretation

Given are two vectors $\mathbf{a}_1$ and $\mathbf{a}_2$.



Project $\mathbf{a}_2$ onto $\mathbf{a}_1$: $\mathbf{v}_1 = (\mathbf{a}_2{}^T\mathbf{a}_1)\mathbf{a}_1$ and $\mathbf{v}_2 = \mathbf{a}_2 - \mathbf{v}_1$.

Normalize: $\mathbf{q}_1 = \dfrac{\mathbf{v}_1}{\|\mathbf{v}_1\|}$ and $\mathbf{q}_2 = \dfrac{\mathbf{v}_2}{\|\mathbf{v}_2\|}$.

computing an orthogonal basis

Input: The k columns of an n-by-k matrix A are $[\mathbf{a}_1\ \mathbf{a}_2\ \cdots\ \mathbf{a}_k]$.
Output: an orthogonal n-by-k matrix $Q = [\mathbf{q}_1\ \mathbf{q}_2\ \cdots\ \mathbf{q}_k]$
with the same column span as A.

The formulas

$$\mathbf{p}_j = \mathbf{a}_j - \sum_{i=1}^{j-1} \mathbf{q}_i(\mathbf{q}_i^T \mathbf{a}_j), \quad \mathbf{q}_j = \frac{\mathbf{p}_j}{||\mathbf{p}_j||_2}, \quad \text{for } j = 1, 2, \ldots, k.$$

are known as the Gram-Schmidt orthogonalization method.

- $\mathbf{p}_j$ is obtained from $\mathbf{a}_j$, subtracting the orthogonal projections of $\mathbf{a}_j$ on the column space of $[\mathbf{a}_1\ \mathbf{a}_2\ \cdots\ \mathbf{a}_{j-1}]$.
- Collecting the coefficients $\mathbf{q}_i^T \mathbf{a}_j$ into r_{ij} leads to an upper triangular matrix R such that $A = QR$, the QR decomposition of A.

the definition of the function

```
function gsqr(A::Array{BigFloat,2})
    k = size(A,2)
    Q = deepcopy(A)
    R = Array{BigFloat,2}(zeros(k,k))
    for j = 1:k
        p = A[:,j]
        for i = 1:j-1
            R[i,j] = transpose(Q[:,i])*A[:,j]
            p = p - Q[:,i]*R[i,j]
        end
        Q[:,j] = p/norm(p,2);
        R[j,j] = transpose(Q[:,j])*A[:,j]
    end
    return Q, R
end
```

formatting numbers in the test

```
using Printf
Base.show(io::IO, f::Float64) = @printf(io, "%.3e", f)
using LinearAlgebra

"""
Generates a random 4-by-3 matrix
and tests the gsqr function
with 64-bit floating-point numbers.
"""
function testBigFloat()
    println("Testing the BigFloat version ...")
    a = rand(4, 3)
    A = Array{BigFloat,2}(a)
    q, r = gsqr(A)
    res = norm(A - q*r)
    stres = @sprintf("%.2e", res)
    println("The residual : $stres")
    println("Testing orthogonality, Q^T*Q is")
    show(stdout, "text/plain", transpose(q)*q); println("");
end
```

exploring the loss of orthogonality

Exercise 2:
The Gram-Schmidt method in its orginal form suffers from a loss of orthogonality.

Setup the following experiment:

1. Run `gsqr` on a sufficiently large random matrix A.
2. Compute the inner product of the last two vectors of the Q, as returned by `gsqr`.

Describe your observations.

Exercise 3:
Repeat the above experiment,
but now on matrices A that are nearly singular.

Introduction to Julia

1. the Language Julia
   - about Julia
   - a first session
2. Exploring Algorithms
   - defining functions
   - cobweb diagrams
3. Fractal Plots
   - the method of Laguerre
   - C, Python, and Julia

fixed-point iterations

Definition

The number $r \in \mathbb{R}$ is *a fixed point* of the function g if $g(r) = r$.

We will reformulate the root finding problem $f(r) = 0$
into a fixed point computation of $g(r) = r$.

Example $f(x) = x^3 + x - 1 = 0$ leads to $g(x) = 1 - x^3 = x$.

A fixed-point iteration applies a simple formula:

$$x_{k+1} = g(x_k), \quad k = 0, 1, \ldots$$

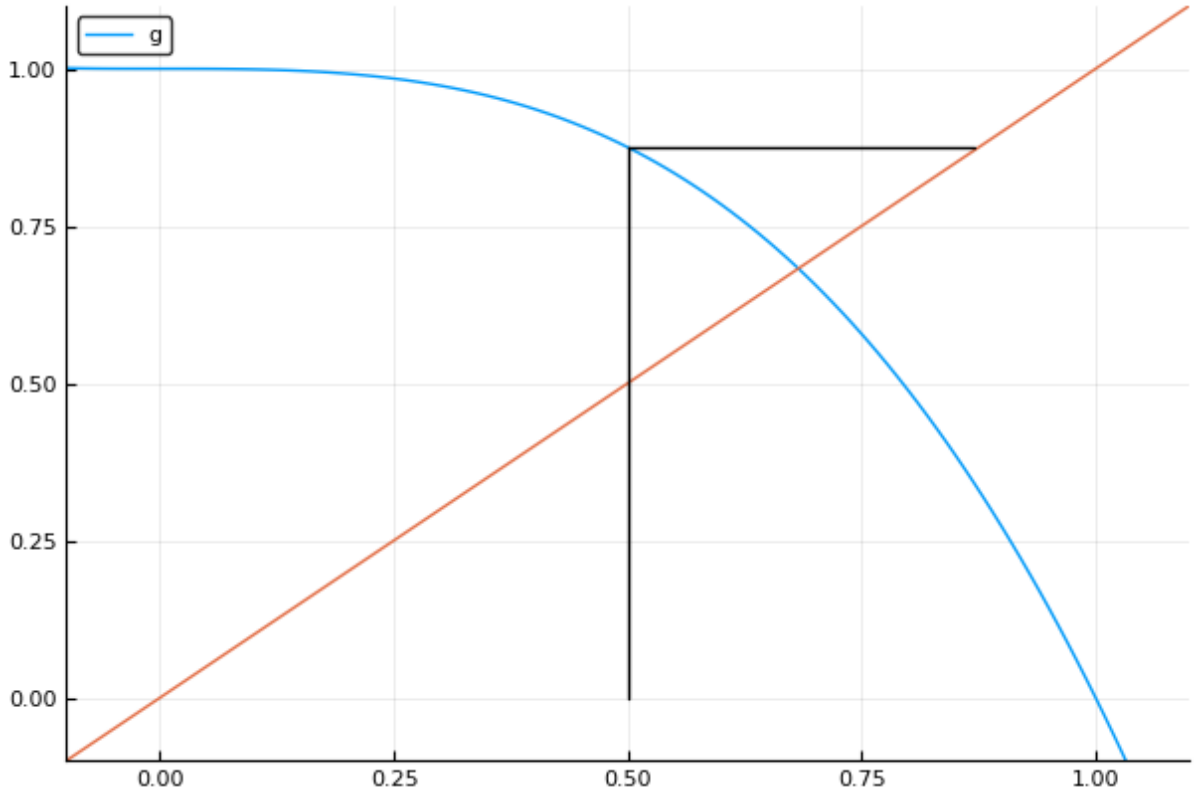
starting at an initial guess x_0 for the fixed point.

If the fixed-point iteration converges, then x_∞ is a fixed point.

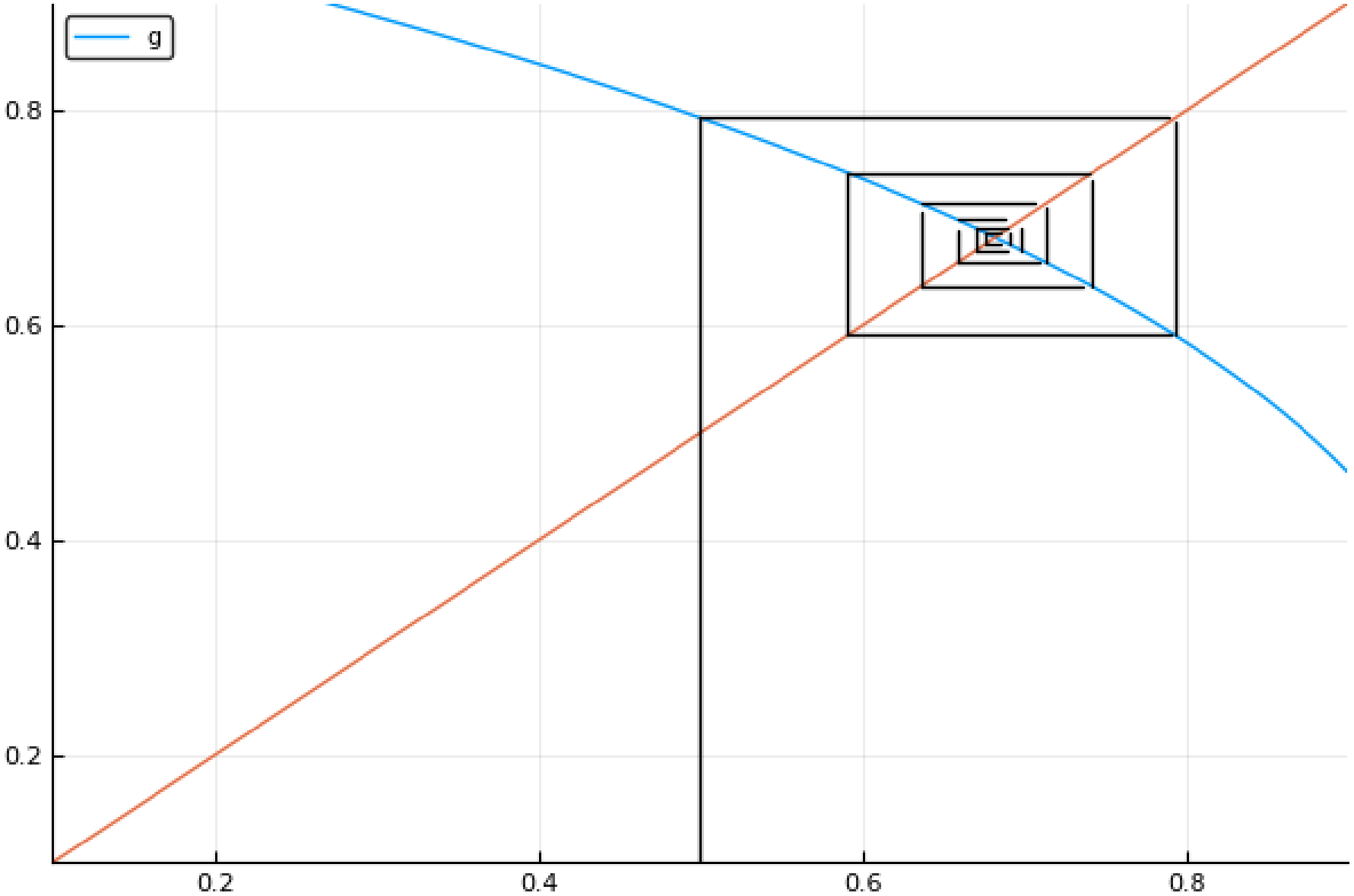
drawing a cobweb diagram

Two steps in executing $x_1 = g(x_0)$:

1. evaluate g: $y = g(x_0)$, draw a vertical line
2. eliminate y: $x_1 = y$, draw a horizontal line



cobweb diagram for $g(x) = \sqrt[3]{1-x}$



plotting instructions for $g(x)$, $x \in [a, b]$

1. plotting $g(x)$ and the diagonal

```
r = a:0.01:b
plt = plot(g, r, label="g")
ylims!((a, b))
diagonal(x) = x
plot!(diagonal, r, label="")
```

2. plotting a line

```
xnext = g(xprevious)
plot!([xprevious], [xnext],
      line=(:sticks, :black), label="")
```

3. etc ...

Introduction to Julia

1. the Language Julia
   - about Julia
   - a first session
2. Exploring Algorithms
   - defining functions
   - cobweb diagrams
3. Fractal Plots
   - the method of Laguerre
   - C, Python, and Julia

the method of Laguerre

The method of Laguerre computes one root of a polynomial.

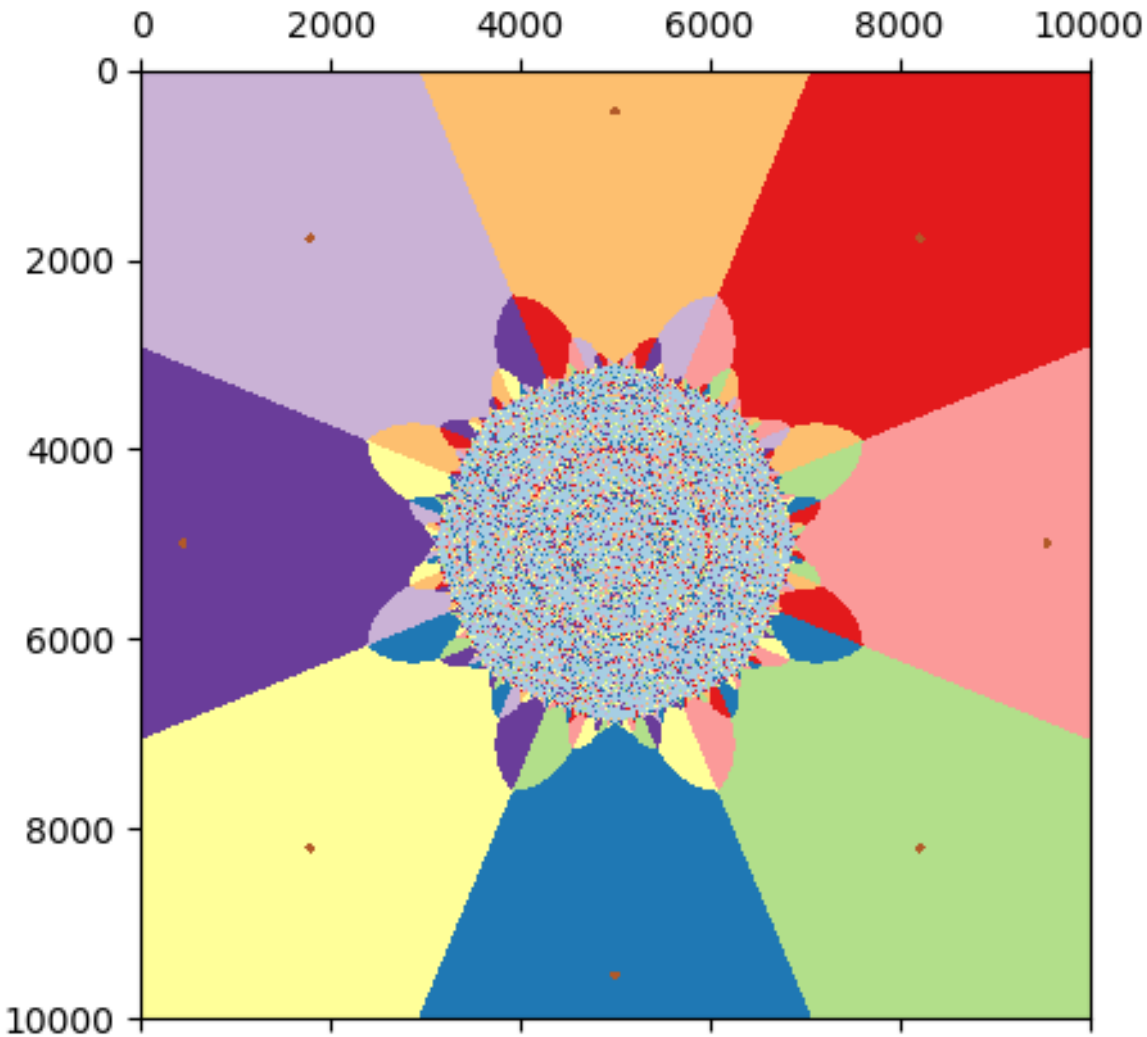
The method is iterative:

1. Applies the quadratic formula.
2. Uses information of the first two derivatives.

Its convergence is cubic to a regular root.

What about the regions of convergence?

basins of attraction of Laguerre's method on $x^8 - 1 = 0$



computational setup

Let $[a, b]$ be the interval for the real parts and the imaginary parts of the point z_0 at which to start the method of Laguerre.

For dimension n, let $\Delta z = (b - a)/(n - 1)$.

Then the grid of points is defined by the loops

for k from 1 to n do
  for ℓ from 1 to n do
    $z_0 = (a + (k - 1)\Delta z) + (a + (\ell - 1)\Delta z)\sqrt{-1}$

For each grid point, we run Laguerre's method and record the number of iterations.

Introduction to Julia

1. the Language Julia
   - about Julia
   - a first session
2. Exploring Algorithms
   - defining functions
   - cobweb diagrams
3. Fractal Plots
   - the method of Laguerre
   - C, Python, and Julia

C, Python, and Julia

The plots are for $a = -1.1$ and $b = +1.1$. For $n = 10001$, the plots require about 100 million calls to the method of Laguerre.

- This takes several minutes for the C code (compiled with `-O3`).
- The Julia program takes about twice as much time.
- Because Python is so much slower, take $n = 501$.

For $n = 10001$, the file with all integers used in the visualization occupies about 200MB.

verify the efficiency on your computer

Exercise 4:

Run the posted `laguerre.c`, `laguerre.py`, and `laguerre.jl` on your computer.

Report the timings on the three versions.

summary

Julia is well suited to explore algorithms.

We can focus on mathematical questions, rather than debugging code.

1. The standard library includes `LinearAlgebra`, `BigFloat`.
2. Programs with the ease of a scripting language.
3. Efficient, close to C.