A (very) brief introduction to Julia

Flemming Holtorf

Computer Science and Artificial Intelligence Laboratory

Department of Chemical Engineering

Massachusetts Institute of Technology

August 2023

What is Julia?

A programming language that challenges notions often treated as "laws of nature"

1. High-level dynamic programs have to be slow.
2. One must prototype in one language and then rewrite in another language for speed or deployment.
3. There are parts of a system appropriate for the programmer, and other parts that are best left untouched as they have been built by the experts.

and it's designed for numerical computing specifically . . .

Solving the Two Language Problem

Two Language Problem

1. Prototype in an easy-to-use high-level (dynamic) language (Python, MATLAB, . . .)

2. Rewrite performance critical components in a hard-to-use low-level (static) language (C, Fortran)

Solutions

- (Vectorization (MATLAB, Python/NumPy, . . .))
- **Dynamic language with fast enough core functionality (for-loops, . . .) that performance critical libraries can be written the language itself**

Solving the Two Language Problem

Two Language Problem

1. Prototype in an easy-to-use high-level (dynamic) language (Python, MATLAB, . . .)
2. Rewrite performance critical components in a hard-to-use low-level (static) language (C, Fortran)

Solutions

- (Vectorization (MATLAB, Python/NumPy, . . .))
- **Dynamic language with fast enough core functionality (for-loops, . . .) that performance critical libraries can be written the language itself**

“We want something as usable for general programming as Python,
as easy for statistics as R, as natural for string processing as Perl,
as powerful for linear algebra as MATLAB,
as good at gluing programs together as the shell.
Something that is dirt simple to learn yet keeps the most serious hackers happy.
We want it interactive, and we want it compiled.
(Did we mention it should be as fast as C?)”

Jeff Bezanson, Stefan Karpinski, Viral B. Shah, & Alan Edelman (2012)

Don’t trust me? Perhaps this may change your mind . . .

- 2019: IEEE Computer Society Sidney Fernbach Award
- 2019: James H. Wilkinson Prize for Numerical Software
- 2018: Best of Open Source Software (Bossie) Award
- 2016: INFORMS Computing Society Prize (for JuMP)
- 2015: IEEE-CS Charles Babbage Award

Goals for today

1. Be exposed to some neat features of the Julia ecosystem
2. Get a rough idea of the concepts that make Julia unique and powerful

   $\Longrightarrow$ **Multiple Dispatch & Type Inference**

3. Revisit and extend some past exercises

But first let's install Julia

1. Download **and install** the current stable release of Julia for your operating system. To that end, follow the instructions at https://julialang.org/downloads/.
2. Download and install VSCode by following the instructions at https://code.visualstudio.com/.
3. Start VSCode and install the Julia extension (`Preferences` → `Extensions`). Type '`julia`' into the searchbar and install the `Julia` extension
4. Restart VSCode.

Exercise 1: Your own cosine approximation

Power series expansion of cosine

$$\cos(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!} \implies \cos(x) \approx \sum_{n=0}^{N} (-1)^n \frac{x^{2n}}{(2n)!}$$

Question 1

Implement a function `cos_approx(x,N)` that implements the power series expansion of cosine evaluated at `x` truncated to the `N`th term and returns the result.

Question 2

Use `@elapsed cos_approx(x,N)` to time how long your function takes to execute for different values of x and N.

Question 3

Go back to your Python code for `cos_approx(x,N)` from last week and time it as well.

Exercise 2: Implement Newton's method in 1D

```
Procedure: 1D Newton's method
1: Initialize guess: $x \leftarrow x_0$
2: while $|f(x)| > \epsilon$ do
3:     $x \leftarrow x - \frac{f(x)}{f'(x)}$
4: end while
5: return x
```

The Power of Abstraction

“Abstraction, which is what good computation is really about, recognizes what remains the same after differences are stripped away.”

- separate what matters structually from problem specific details (readability)
- translate insights gained on a low level to various settings with minimal effort (reusability)

Abstraction + Specialization: Explicit Runge-Kutta methods

Algorithm

Successively evaluate:

$$
\begin{aligned}
k_1 &= f(x(t), t) \\
k_2 &= f(x(t) + h a_{2,1} k_1, t + c_2 h) \\
&\vdots \\
k_s &= f\left(x(t) + h \sum_{j=1}^{s-1} a_{s,j} k_j, t + c_s h\right)
\end{aligned}
$$

Update: $\boxed{\hat{x}(t+h) = x(t) + h \sum_{i=1}^{s} b_i k_i}$

Method Data

$$
\begin{array}{c|cccc}
c_1 & & & & \\
c_2 & a_{2,1} & & & \\
\vdots & \vdots & \ddots & & \\
c_s & a_{s,1} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}
$$

Abstraction + Specialization: “Mundane” Examples

Addition

- Numbers

$$37 + 5 = 42$$

- Functions: Given $f, g : \mathbb{R} \to \mathbb{R}$

$$(f+g) : \mathbb{R} \to \mathbb{R} \text{ such that } (f+g)(x) = f(x) + g(x) \text{ for all } x \in \mathbb{R}$$

- Matrices: Given $A, B \in \mathbb{R}^{n \times m}$

$$A + B = \begin{bmatrix} A_{11} + B_{11} & \cdots & A_{1N} + B_{1N} \\ \vdots & \ddots & \vdots \\ A_{N1} + B_{N1} & \cdots & A_{NN} + B_{NN} \end{bmatrix}$$

Abstraction + Specialization: "Mundane" Examples

Multiplication

- Numbers

$$14 \cdot 3 = 42$$

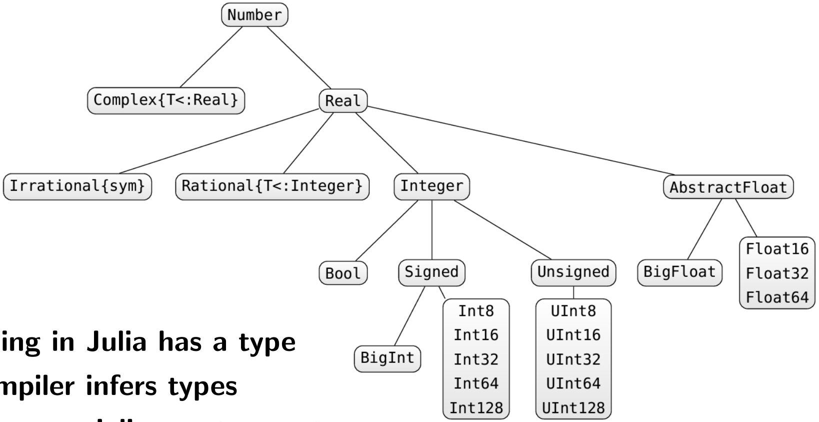
- Functions: Given $f, g : \mathbb{R} \to \mathbb{R}$

$$(f \cdot g) : \mathbb{R} \to \mathbb{R} \text{ such that } (f \cdot g)(x) = f(x) \cdot g(x) \text{ for all } x \in \mathbb{R}$$

- Matrices: Given $A, B \in \mathbb{R}^{n \times n}$

$$A \cdot B = C \text{ such that } C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Abstraction via **Multiple Dispatch** & **Type Inference**



- **Everything in Julia has a type**
- **The compiler infers types**
- **Functions specialize on types at compile time**

Exercise 3: Simplified parameter inference

1D Heat equation

$$\lambda \frac{\partial^2 T}{\partial x^2} + Q = 0$$

Spatial discretization $\Longrightarrow$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 & 0 \\ \vdots & & \ddots & \ddots & \ddots & & \vdots \\ 0 & 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} T_0 \\ T_1 \\ T_2 \\ \vdots \\ T_{N-1} \\ T_N \\ T_{N+1} \end{bmatrix} = \begin{bmatrix} T_{\text{lower}} \\ -\frac{\Delta x^2}{\lambda} Q_0 \\ -\frac{\Delta x^2}{\lambda} Q_1 \\ \vdots \\ -\frac{\Delta x^2}{\lambda} Q_{N-1} \\ -\frac{\Delta x^2}{\lambda} Q_N \\ 0 \end{bmatrix}$$

We want to find λ so that $T(500\textbf{km}) = 640K$ with Newton's method.

Steps

1. Solve the 1D heat equation in Julia (**see syntax hints on the next page**)
   1.1 Create three vectors that carry the entries of the three diagonals
   1.2 Create the triadiagonal coefficient matrix with `Tridiagonal`
   1.3 Solve the tridiagonal system with $\backslash$
   1.4 Plot the result
2. Use your solving strategy to implement a function that computes the difference between $T(500\text{km})$ and our target temperature of 640K as a function of λ.
3. Use automatic differentiation to compute the derivative of this function (**we will do that together!**)
4. Use your Newton method implementation to solve for λ so that $T(500\text{km}) = 640\text{K}$.

Hints

- You can use `zeros(N)` to create a vector of length N with all zeros
- You can use `ones(N)` to create a vector of length N with all ones
- You can use the `Tridiagonal(a,b,c)` function to create a tridiagonal matrix

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & 0 & 0 & 0 \\ a_1 & b_2 & c_2 & 0 & 0 & 0 & 0 \\ 0 & a_2 & b_3 & c_3 & 0 & 0 & 0 \\ \vdots & & \ddots & \ddots & \ddots & & \vdots \\ 0 & 0 & 0 & a_{N-3} & b_{N-2} & c_{N-2} & 0 \\ 0 & 0 & 0 & 0 & a_{N-2} & b_{N-1} & c_{N-1} \\ 0 & 0 & 0 & 0 & 0 & a_{N-1} & b_N \end{bmatrix}$$

- To solve a linear equation $Ax = b$ for x, you can use `A\b`
- To plot a curve, you can use `plot(x,y)` where `x` and `y` are vectors with the x- and y-coordinates, respectively.

Exercise 4: Time-dependent heat equation as IVP

If we discretize the PDE

$$\frac{\partial T}{\partial t} = \lambda \nabla^2 T + Q(t)$$

only spatially, we end up with an IVP:

$$\frac{d}{dt}\begin{bmatrix} T_1(t) \\ T_2(t) \\ T_3(t) \\ \vdots \\ T_{N-1}(t) \\ T_N(t) \end{bmatrix} = \frac{\lambda}{\Delta x^2}\begin{bmatrix} T_2(t) - 2T_1(t) + T_{\text{lower}} \\ T_3(t) - 2T_2(t) + T_1(t) \\ T_4(t) - 2T_3(t) + T_2(t) \\ \vdots \\ T_N(t) - 2T_{N-1}(t) + T_{N-2}(t) \\ -T_N(t) + T_{N-1}(t) \end{bmatrix} + \begin{bmatrix} Q_1(t) \\ Q_2(t) \\ Q_3(t) \\ \vdots \\ Q_{N-1}(t) \\ Q_N(t) \end{bmatrix}, \quad \begin{bmatrix} T_1(0) \\ T_2(0) \\ T_3(0) \\ \vdots \\ T_{N-1}(0) \\ T_N(0) \end{bmatrix} = 200\,\text{K}$$

We learned how to solve IVPs ... and we can do it even more easily in Julia!

Intruiged about Julia? There are a ton of resources out there to learn more

- Why Julia was created
- Julia documentation: Noteworthy differences to other common languages
- Julia for data science
- Lot's of learning resources

Thank you!

github.com/FHoltorf

holtorf@mit.edu

References