

A (very) brief introduction to Julia

Flemming Holtorf

Computer Science and Artificial Intelligence Laboratory

Department of Chemical Engineering

Massachusetts Institute of Technology

July 2022



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

Solutions

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

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
- ▶ 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

⇒ **Multiple Dispatch & Type Inference**

3. Implement your own mini automatic differentiation code

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

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.

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 structurally from 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:

$$k_1 = f(x(t), t)$$

$$k_2 = f(x(t) + ha_{2,1}k_1, t + c_2h)$$

\vdots

$$k_s = f(x(t) + h \sum_{j=1}^{s-1} a_{s,j}k_j, t + c_sh)$$

Update:

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

Method Data

c_1				
c_2	$a_{2,1}$			
\vdots	\vdots	\ddots		
c_s	$a_{s,1}$	\cdots	$a_{s,s-1}$	
	b_1	b_2	\cdots	b_s

Abstraction + Specialization: “Mundane” Examples

Addition

- ▶ Numbers

$$37 + 5 = 42$$

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

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

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

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

Abstraction + Specialization: “Mundane” Examples

Multiplication

- ▶ Numbers

$$14 \cdot 3 = 42$$

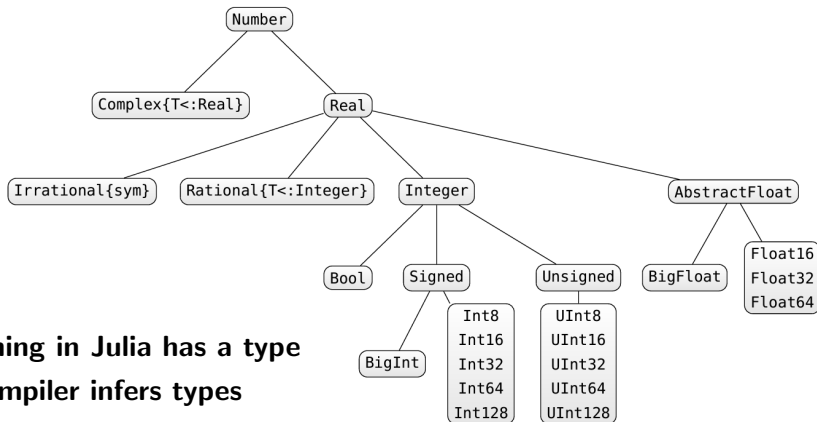
- ▶ Functions: Given $f, g : \mathbb{R} \rightarrow \mathbb{R}$

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

- ▶ 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 2

Question 1

Implement a function `my_add` that takes in two elements of type `MyComplex` and returns their sum as object of `MyComplex` type.

Question 2

Define a new method for the `Base.+` function for `MyComplex` types. Then call

```
sum([MyComplex(1.0, 2.0), MyComplex(3.0, 4.0)])
```

Does it work? Did you expect that?

Forward Mode Automatic Differentiation

Idea: Propagate derivatives through compositions of functions by applying the rules of differentiation

Dual Number (nilpotent) Arithmetic

- ▶ Dual number: $x + y\epsilon$ with $x, y \in \mathbb{R}$.
- ▶ First order Taylor approximation maps dual numbers to dual numbers

$$f : \underbrace{x + \epsilon}_{\text{dual number}} \rightsquigarrow \underbrace{f(x) + f'(x)\epsilon}_{\text{dual number}}$$

- ▶ Rules of differentiation defines an arithmetic for dual numbers:

$$(f + g) : x + \epsilon \rightsquigarrow f(x) + g(x) + [f'(x) + g'(x)] \epsilon \quad (\text{Addition Rule})$$

$$(f \cdot g) : x + \epsilon \rightsquigarrow f(x)g(x) + [f(x)g'(x) + f'(x)g(x)] \epsilon \quad (\text{Product Rule})$$

$$(f \circ g) : x + \epsilon \rightsquigarrow f(g(x)) + f'(g(x))g'(x)\epsilon \quad (\text{Chain Rule})$$

Exercise 3

Question 1

Complete the provided template `dual_number_arithmetic.jl` by using the rules of differentiation.

Question 2

Use dual number arithmetic to get the derivative of Simone's test function

$$f(x) = x \cos(x) + \sin(x)$$

To that end, complete the template `simones_test_function.jl`.

Exercise 3

Question 3

Consider the Lotka-Volterra model:

$$\frac{d}{dt} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_1 - x_1 x_2 \\ -2x_2 + \alpha x_1 x_2 \end{bmatrix}, \quad \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 1.0 \\ 0.25 \end{bmatrix}$$

Use dual number arithmetic to evaluate the sensitivities $\frac{dx_1(15)}{d\alpha}$, $\frac{dx_2(15)}{d\alpha}$. To that end, complete the template `lotka_volterra.jl`.

Intrigued 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