Wissenschaftliches Programmieren

Meik Hellmund

Meik.Hellmund@math.uni-leipzig.de

https://www.math.uni-leipzig.de/~hellmund/teaching/julia.html

- Montags 11:15 HS 5
- A-Woche Donnerstag 13:15 HS 5



Organisatorisches

- Pflichtmodul `10-MAT-BM006` für Bachelor Mathematik, 3 SWS, 5 CP, ohne Benotung
- Bitte anmelden unter: `https://miserv101.mathematik.uni-leipzig.de/registration/r/`
  invitation code: `LeiXP2427` ⇒ Zugangsdaten für JupyterLab-Server

Links zum Kurs

- Moodle `https://moodle2.uni-leipzig.de/course/view.php?id=62050`
- JupyterLab-Server `https://misun103.mathematik.uni-leipzig.de`

Ihr Job

- **Übungsblätter:** etwa 14-tägig, fakultativ (keine Prüfungsvorleistung), Besprechung an den Donnerstag-Terminen
- **Prüfungsleistung** (ohne Benotung): Ein Programmierprojekt bestehend aus
  - der Bearbeitung einer Programmieraufgabe in Julia,
  - einer schriftlichen Ausarbeitung zu dieser Aufgabe in Form einer kleinen wissenschaftlichen Arbeit in englischer Sprache unter Verwendung von LaTeX und
  - einer Präsentation in englischer Sprache unter Verwendung von LaTeX.

Course content (preliminary)

Scientific Computing

- Programming language: Julia
  - Basics, syntax
  - Ecosystem: package management, REPL, introspection
  - Computer arithmetic, machine numbers
  - Type system
  - Arrays and broadcasting
  - Plots
  - Scientific programming
  - Data structures & algorithms

Scientific Writing

- Document markup language: LaTeX
  - Introduction to LaTeX fundamentals
  - Math into LaTeX
  - References, citations, and automatic labelling
  - Figures, tables, floats
  - Typography, fonts
  - Presentations with the `beamer` class

If you want to learn, AI is a fabulous tool.

If you don't want to learn, AI is a fabulous tool.

Why Julia?

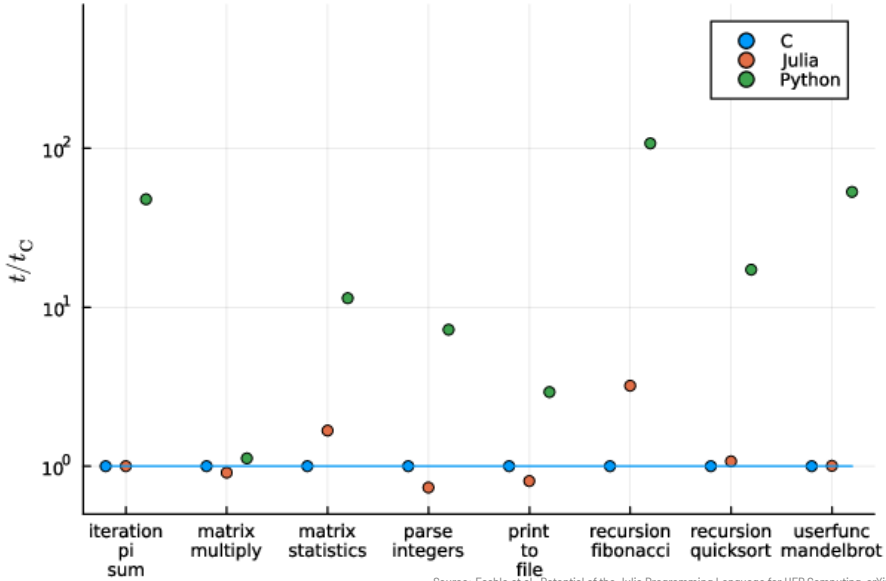
"Julia is an open-source, multi-platform, high-level, high-performance language for technical computing."

- Developed for *scientific computing*, from laptops to clusters.
- Core goal: speed of C & ease of Python/MATLAB.
- Intuitive, compact, modern syntax.
- Generic, composable, and reusable code.
- Just-in-time (JIT) compilation supports interactive development.
- Large ecosystem, integrated package manager.
- Open source and open development.

- Started 2009 at the Computer Science and Artificial Intelligence Laboratory, MIT
- 2012 first release version 0.1, blog article Why we created Julia
- 2018 version 1.0
- current: version 1.13.1 (2026-09-25)

Click for a picture: the creators of Julia, Stefan Karpinski, Viral Shah, Jeff Bezanson, and Alan Edelman

Lies, damned Lies, & Benchmarks



Source: Eschle et al., Potential of the Julia Programming Language for HEP Computing, arXiv:2306.03675, CC BY 4.0

Calculate $12^{1/3} + \dfrac{3\sqrt{2}}{\sin(0.5) - \cos(\frac{\pi}{4})\log(3)} + e^5$

```
julia> 12^(1/3) + 3sqrt(2)/(sin(.5) - cos(π/4)*log(3)) + exp(5)
136.43732662344087
```

- Exponents are denoted using the caret symbol: `a^b`.
- Predefined constants such as π are available.
- The function `log()` denotes the natural logarithm by default.
- The multiplication operator $*$ may be omitted when a number is followed by a variable, function, or opening parenthesis.
- `x ÷ y` is integer division, rounded to zero; and `x % y` the corresponding remainder:

```
julia> 14 ÷ -3
-4
julia> 14 % -3
2
```

A short note about Unicode

- Julia uses *Unicode*, providing over 170 scripts plus many mathematical and technical symbols (and over 3000 emojis).
- So, `ΓΩ`, `🎷🕺`, `текст`, `äääåâh`, `גבא`, `𝔐ℝ𝒱` are all valid names for variables, functions etc.
- Julia also understands some Unicode operators and symbols. For instance, use

  `≤`, `≠`, `≥`, `π`, `∈`, `√`, `a÷b` instead of `<=`, `!=`, `>=`, `pi`, `in`, `sqrt`, `div(a,b)`.
- Julia allows the input of thousands of Unicode characters via a LaTeX-like syntax with tab completion.
  - `\alpha` ⇆ generates α.
  - `\euler` ⇆ generates e – the Euler number `exp(1)`.
  - `\le` ⇆ generates ≤.
  - `\in` ⇆ generates ∈.
  - `\:rainbow:` ⇆ generates 🌈.
- See https://docs.julialang.org/en/v1/manual/unicode-input/ for the full list.

A real code example

```
function T(θ::AbstractArray, 𝒞::Tuple{AbstractArray, Vararg{AbstractArray}},
    𝒟::Tuple{AbstractArray, Vararg{AbstractArray}})
⊗ = kron
l = length(θ)
Il = SMatrix{l,l}(1.0I)
T = @SMatrix zeros(l,l)
N = length(𝒟[1])
ℳ, ℳ′ = 𝒟
Λ1, Λ2 = 𝒞
Λn = @MMatrix zeros(4,4)
e1 = @SMatrix [1.0; 0.0; 0.0]
for n = 1:N
    Λn[1:2,1:2] .=  Λ1[n][index,index]
    m′= hom(ℳ′[n])
    Un = (m ⊗ m′)
    ∂xun =  [(e1 ⊗ m′) (e2 ⊗ m′) (m ⊗ e1) (m ⊗ e2)]
    Bn =  ∂xun * Λn * ∂xun'
    Σn = θ' * Bn * θ
    Σn^-1 = inv(Σn)
    ...
```

Source: https://juliamono.netlify.app/

Variables & Assignments

- A *variable* is a name bound to a *value*.
- A variable is created or re-bound by assignment:

```
julia> x =  2.3 + 1.2
3.5
julia> x = "Leipzig"
"Leipzig"
```

Semantics of the assignment operator "="

- Evaluate the right-hand side (RHS).
- If the left-hand side (LHS) is a variable name, bind the variable to the computed value.
- If the LHS refers to a field of a *mutable struct* or an element of a *mutable container*, modify the contents of the struct/container.
- The assignment operation itself evaluates to the RHS value, enabling chained assignments:

```
julia> a = (b = 2+2) + 3
7
julia> b
4
```

First Aid

- Use the Julia REPL (read-eval-print loop) for experiments!
- Use the ⇆ key and the ? key!

```
julia> lo⇆
local           lock
log             log10
log1p           log2
logrange        lowercase
lowercasefirst
julia> pri⇆
primitive type
print
println
printstyled
```

```
julia>?for
  for loops repeatedly evaluate a block of
  statements while iterating over a
  sequence of values.
  ...
  Example

  julia> for i in [1, 4, 0]
             println(i)
         end
  1
  4
  0
```

Control flow

```
println("🎮 FizzBuzz Game (1-20):")
for i ∈ 1:20
    if i % 3 == 0 && i % 5 == 0
        print("$i -> FizzBuzz!")
        println(" 🎉")
    elseif i % 3 == 0
        println("$i -> Fizz")
    elseif i % 5 == 0
        println("$i -> Buzz")
    else
        println("$i")
    end
end
```

- `if <condition> <statement> end`
- zero or more `elseif` branches
- zero or one final `else` branch (catch-all)

```
year = 2020
if year % 4 == 0 && (year % 100 ≠ 0 ||
        !(year % 400 ≠ 0))
    println("$year is a leap year ✓")
end
```

- Tests `<,>,==,!=,>=,<=,===`
- Logical operators `&&,||,!`
- String interpolation with `$`
- Simplest loop:
  `for k ∈ first:last ... end`

Syntax: statements and comments

- Line breaks separate statements **unless** the parser sees an unfinished expression.
- Semicolons separate multiple statements on one line.
- In the REPL/jupyter notebook, a final semicolon suppresses output.
- Indentation is optional but *highly* recommended!
- Everything from a `#` to the end of line is a comment.
- Everything between `#=` and `=#` is a comment. Multi-line comments `#= ... =#` can be nested like parentheses.

```
julia> x = 2   # no, let x = 3
2
julia> x #= a rare variable name =# = 3
3
```

Syntax: blocks

- A block is a sequence of statements introduced by a keyword.
- Common keywords: **`if`**, **`for`**, **`while`**, **`function`**, **`begin`**.
- Indentation is optional but recommended.
- A simple block can be a one-liner:
  `if x > 0 flag = 1 end`

```
function classify(x)
    if x > 0
        kind = "positive"
    elseif x < 0
        kind = "negative"
    else
        kind = "zero"
    end
    kind
end

for v in [1, 0, -2]
    println(classify(v))
end
```

Functions

```
function hyp(a, b)
    if a == 0 return b end
    if b == 0 return a end
    c² = a^2 + b^2
    return sqrt(c²)
end


# one-liner variant
hyp_1(a,b) = sqrt(a^2 + b^2)


# named anonymous variant
hyp_a = (a,b) -> sqrt(a^2+b^2)
```

- A function returns the last expression by default.
- `return` is optional for the final value.
- Short functions can be one-liners.

```
julia> hyp(3, 4)
5.0
```

1-dim Arrays (Vectors)

Julia arrays are written and indexed with square brackets. Indexing **starts with 1!**

```
julia> v = [22, 3.4, -7, 8/9, sin(2)]
5-element Vector{Float64}:
 22.0
  3.4
 -7.0
  0.8888888888888888
  0.9092974268256817
julia> v[2] = v[4]; v
5-element Vector{Float64}:
 22.0
  0.8888888888888888
 -7.0
  0.8888888888888888
  0.9092974268256817
```

```
julia> v = []
julia> push!(v, 33)
julia> push!(v, 111)
julia> v
2-element Vector{Any}:
  33
 111
julia> v[1]
33.0
julia> v[end]
111.0
```

Often useful: Create an empty vector and add elements.