

Introduction to Julia

Markus Kuhn

Department of Computer Science and Technology University of Cambridge

https://www.cl.cam.ac.uk/teaching/2425/TeX+Julia/

Technical computing languages

- support rapid prototyping and interactive exploration of numerical algorithms and data sets
- high-level language (garbage collecting, var-len structures)
- comprehensive support for linear algebra, statistics, and plotting
- main data types: multi-dimensional numeric arrays and matrices
- most operators and functions work on entire matrices
 rarely necessary to write out loops
- use internally highly optimized numerical libraries (BLAS, LAPACK, FFTW)
- support interactive use via read-evaluate-print loop (REPL)
- interpreted or just-in-time compiled, notebook support
- comprehensive toolboxes/modules/packages for easy access to standard algorithms from many fields: statistics, machine learning, image processing, signal processing, neural networks, wavelets, communications systems, etc.
- very easy I/O for many data/multimedia file formats
- widely used as a visualization and teaching tool

Older contenders

MATLAB – "matrix laboratory" student tool (University of New Mexico, 1970s), commercial product since 1984, very widely used since 1990s in engineering simulations and teaching, initially not a general-purpose language (e.g., object classes only from 2008, dictionaries only added in 2022), integrated IDE, since 2000 based on Java JVM code generation

Campus licence: https://uk.mathworks.com/academia/tah-portal/the-university-of-cambridge-666637.html

- Similar to (earlier versions of) MATLAB, subset compatible: GNU Octave, SciLab, FreeMat
- R focus on statistics and plotting https://www.r-project.org/
- **Python** a full-featured programming language. Modules:
 - numpy MATLAB-like numerical arrays, fast linear algebra
 - matplotlib MATLAB-like plotting functions https://matplotlib.org/
 - SciPy scientific computing, Pandas data analysis, etc.

• others: LuaJIT (SciLua), Perl Data Language (PDL), OCaml (Owl) R and especially Python became very popular \approx 2000–2010, but are much slower than compiled statically typed C/C++/Fortran, as their dynamic types were intended for interpreted execution. In HPC applications, they remain mainly configuration/glue languages for C/C++/Fortran libraries.







3 / 28

Julia

Modern, fast, full-featured, compiled, interactive language, initially created 2009–2012 at Massachusetts Institute of Technology by J. Bezanson, A. Edelman, S. Karpinski, V.B. Shah.

 MATLAB-inspired syntax (especially much nicer and compacter array syntax than NumPy)



- not intended to be MATLAB compatible
- combines dynamic and static type systems via multiple dispatch
- just-in-time compiled by LLVM backend
- with some care, Julia code can execute nearly as fast as C/C++
- aims to solve the two-language problem (versus e.g. Python having to call C/C++ code for performance)
- can also call C, C++, Python, R, Fortran functions
- LISP-like metaprogramming, rich flexible parametric type system
- built-in package manager for easy access to package ecosystem, version-controlled virtual package environments ("projects")
- multiple dispatch helps with reusing types across packages
- Backwards-compatible since version 1.0 (2018)

Julia shortcomings

start-up delay when loading/calling large packages, as they need to be JIT compiled first ("time to first plot")

Since Julia 1.9 much faster thanks to on-disc cache for pre-compiled object code of packages.

not aimed at compilation of stand-alone binaries

Possible with PackageCompiler.jl, but requires extra effort and can lead to large (> 100 MB) binaries.

not aimed at hard real-time applications: heap memory allocation and automatic mark-and-sweep garbage collection can introduce non-deterministic delays

Manual control of memory allocation is possible, but not typical Julia style, and often not supported by ecosystem packages.

- package ecosystem and package documentation sometimes still less mature or complete than that of Python, R, MATLAB But it is very easy to get involved via Pkg.develop and GitHub pull requests.
- diversity of package ecosystem can be confusing initially (e.g. several competing major plotting libraries), usually more than one way of doing everything (esp. compared to MATLAB) Hence this guick introductory tour!

Download the current stable release (e.g. v1.10.5) from: https://julialang.org/downloads/

Windows: Run the 64-bit installer (e.g., julia-1.10.5-win64.exe), then add "C:\Program Files\Julia-1.10\bin" to your PATH environment variable. Also: install Windows Terminal and call Julia inside that, for much better terminal-emulation behaviour than in cmd.exe.

Homebrew: (on macOS or Linux)

```
$ brew install --cask julia
```

or

macOS: Install the 64-bit .dmg package, then add to your PATH the path "/Applications/Julia-1.10.app/Contents/Resources/julia/bin".

Linux: Download the julia-1.10.5-linux-x86_64.tar.gz tarball and unpack somewhere convenient, e.g. at /opt/julia-1.10.5 with e.g.

\$ sudo bash

cd /opt && tar xvzf /path/to/julia-1.10.5-linux-x86_64.tar.gz Then add "/opt/julia-1.10.5/bin" to your PATH environment variable.

Installing Julia via Juliaup

Juliaup is an installation manager that automates installing and updating Julia. It is particularly convenient if you want to have several versions of Julia installed simultaneously.

Follow the instructions at

```
https://julialang.org/downloads/
```

Windows: install Julia with Juliaup using either

Microsoft Store

C:\>winget install julia -s msstore

Also: install Windows Terminal and call Julia inside that, for much better terminal-emulation behaviour than in cmd.exe.

Linux/Unix/macOS:

Run in your terminal the shell command line

```
$ curl -fsSL https://install.julialang.org | sh
```

This will interactively guide you through installing juliaup and a julia wrapper command in ~/.juliaup/bin/ and help you to add that folder to your PATH environment variable.

The Julia documentation at

```
https://docs.julialang.org/
```

consists of the core language manual, plus the reference manuals for

- "Base" the built-in standard functions and types
- "Standard Library" packages preinstalled with Julia

These reference manuals are autogenerated from "docstrings" embedded in the source code. You can also read these docstrings from the REPL help mode with "?function".

Most of "Base" and "Standard Library" are written in Julia, with some C. The @less macro followed by a function call displays the called method in the Julia source code, e.g.

```
julia> @less exit()
```

```
exit() = exit(0)
```

shows that exit() just calls exit(0), while providing an integer exit code calls the internal C function jl_exit:

```
julia> @less exit(0)
exit(n) = ccall(:jl_exit, Cvoid, (Int32,), n)
```

Basic REPL use

Invoking julia without a script.jl filename prints a banner and starts the REPL. Enter Julia expressions and it will display the result:

\$ julia

```
_ _ _(_)_ | Documentation: https://docs.julialang.org
(_) | (_) (_) |
_ _ _ | | _ __ | Type "?" for help, "]?" for Pkg help.
| | | | | | / _` | |
| | _ | | | | (_| | Version 1.10.5 (2024-08-27)
_/ |\__'_|_||\__'_| Official https://julialang.org/ release
|__/ |
julia> 6*7
```

Press one of the keys ?] ; to switch the julia> REPL prompt into one of these alternate REPL modes: help mode, package manager, shell mode:

```
help?> exit
(@v1.10) pkg> status
shell> date
```

Press backspace to leave each mode and return to the julia> prompt. Type Ctrl-D or exit() to leave julia.

Installing packages

Julia's Standard Library does not contain e.g. plotting, audio or digital-signal-processing functions, but add-on packages that provide these (and their dependencies) can be installed easily from the REPL. Hit the] key to enter pkg mode, then type e.g.

```
(@v1.10) pkg> add Plots WAV DSP
(@v1.10) pkg> status
Status `/home/mgk25/.julia/environments/v1.10/Project.toml`
[717857b8] DSP v0.7.10
[91a5bcdd] Plots v1.40.8
[8149f6b0] WAV v1.2.0
```

Julia's Pkg manager downloads https://github.com/JuliaRegistries/General and searches in it for the latest versions of the registered packages you asked to add. [You can also ask for specific versions (add Plots@1.35.3) or add unregistered packages by providing a git URL.]

To modify a downloaded package, use e.g. "dev WAV" to prepare and use a local git clone of that package in ~/.julia/dev/WAV/. Later use "free WAV" to return to a registered version.

Quick docs: type ? or prefix a pkg-mode command with ? Full Pkg.jl documentation: https://pkgdocs.julialang.org/ Packages and their metadata are all installed into ~/.julia/ by default. Set the environment variable JULIA_DEPOT_PATH if you want them elsewhere.

Julia basic types and their literals

```
Bool
                           false, true
Int, Int8, ..., Int128
                           123, 1_000_000, UInt128(2)^127
UInt, UInt8, ..., UInt128 0xff, 0x0012, 0b1011, 0o377
Float64, Float32, Float16 .5, 1.0, 3e6, 2.3f9, NaN, -Inf16
Complex{Float64}
                           0.0 + 1.0im
Rational{Int64}
                           3//4 + 1//2 == 5//4
                           'a', '\n', '\u20ac'
Char
String
                           "hi", "I am \"$name\"", "1+1=$(1+1)"
Symbol
                           :test
Vector{Int} = Array{Int,1} [1, 2, 3]
Matrix{Int} = Array{Int,2} [1 2; 3 4]
Tuple{Int64,Char,Bool} (1, 'a', false)
Nothing
                           nothing
Missing
                           missing + 1 == missing
```

Int and UInt arithmetic is not checked for overflow, like in C: $2^{64}=0$ Use floating-point literals to get floating-point operations: $2.0^{64} > 0$ Type aliases on 64-bit CPUs: Int = Int64, UInt = UInt64

Julia matrices

```
Assign a 3 × 3 matrix of integers:

julia> a = [8 1 6; 3 5 7; 4 9 2]

3×3 Matrix{Int64}:

8 1 6

3 5 7

4 9 2
```

Semicolons equal line feeds:

```
julia> a = [8 1 6
3 5 7
4 9 2]
3×3 Matrix{Int64}:
8 1 6
3 5 7
4 9 2
```

Access a single element:

```
julia> a[2,3]
7
```

Vector and matrix indices start at 1. The first index selects the row, the second the column, like in linear algebra notation.

```
julia> a[3,2];
```

```
julia> ans
9
```

The REPL normally prints the value returned by each expression entered (assignment returns the value assigned). Following an expression with a semicolon suppresses this.

The value of the last expression evaluated in the REPL is also assigned to variable ans.

Vectors are one-dimensional arrays that act in a linear-algebra context like a vertical/column vector:

```
julia> b = [1, 2, 3]
3-element Vector{Int64}:
1
2
3
```

They are different from horizontal/row vectors, which are stored as two-dimensional $\mathbf{1}\times n$ matrices:

```
julia> b = [10 20 30]
1×3 Matrix{Int64}:
10 20 30
```

Julia range objects

start:stop and start:step:stop generate a range of numbers:

```
-1:3 == [-1, 0, 1, 2, 3]

3:0 == Int64[]

1:3:12 == [1, 4, 7, 10]

3:-0.5:1 == [3.0, 2.5, 2.0, 1.5, 1.0]
```

The colon actually generates a range object, which behaves like a vector when used like one. The collect function copies that emulated vector into a real vector in memory.

```
Loop example:

julia> b = 0; for i in 1:10; b += i; end; b

55

Alternatively:

range(1, length=10) == 1:10

range(1, step=2, stop=10) == 1:2:10
```

Vectors and ranges as matrix indices select several rows and columns. When used inside a matrix index, the variable end provides the highest index value: a[end, end-1] == 9. Using just ":" is equivalent to "1:end" and can be used to select an entire row or column.

Row and column selection

Select rows, columns and submatrices of a julia> a[:,:] 3×3 Matrix{Int64}: 1 8 6 3 5 7 4 9 2 julia> a[1,:] 3-element Vector{Int64}: 8 1 6 julia> a[:,1] 3-element Vector{Int64}: 8 3 4 julia> a[2:3,1:2]

```
2×2 Matrix{Int64}:
3 5
4 9
```

```
Matrices can also be accessed as a
1-dimensional vector
julia> a[1:5]
5-element Vector{Int64}:
 8
 3
 4
 1
 5
julia> a[6:end]
4-element Vector{Int64}:
 9
 6
 7
 2
julia> a[1:4:9]
3-element Vector{Int64}:
 8
 5
 2
Julia matrices use column-major storage
order, like Fortran/MATLAB/R, unlike C.
```

Element-wise operators and broadcasting

 $\label{eq:prefix} \mbox{ any operator with } . \mbox{ to apply it element-by-element to matrices and } vectors. For element-wise function calls, insert dot before opening parenthesis. \\$

```
julia> [1 2 3] + 5
ERROR: MethodError:
For element-wise addition, use
broadcasting with dot syntax:
array .+ scalar
julia> [1 2 3] .+ 5
1×3 Matrix{Int64}:
6 7 8
julia> 2 .^ [1 2 3]
1×3 Matrix{Int64}:
2 4 8
julia> sqrt.([4 9 16])
1×3 Matrix{Float64}:
2.0 3.0 4.0
```

Dotted operators also grow (broadcast) vectors and matrices along singleton dimensions, until both operands have the same dimensions:

```
julia> [8 1 6; 3 5 7] .+ [10; 20]
2×3 Matrix{Int64}:
    18 11 16
    23 25 27
```

Combining matrices and vectors

Use [] to build new matrices, where ; joins submatrices vertically (dimension 1), space (or ;;) joins them horizontally (dimension 2), ;;; joins them in dimension 3, etc. The , does *not* join matrices or vectors, it separates elements.

```
julia> a = [8 1 6; 3 5 7; 4 9 2] julia> [[1,2],[3,3]]
3×3 Matrix{Int64}:
                                 2-element Vector{Vector{Int64}}:
                                  [1, 2]
8 1 6
                                  [3, 3]
3 5 7
4 9 2
                                 You can also mask elements:
                                 julia> a .> 5
julia> d = [a[:,end] a[1,:]]
                                 3×3 BitMatrix:
3×2 Matrix{Int64}:
                                  1 0 1
6 8
                                  0 0 1
7 1
                                  0 1 0
2 6
                                 julia> a[a .> 5] .= 0 ; a
julia> e = [zeros(1,3); a[2,:]']
                                 3×3 Matrix{Int64}:
2×3 Matrix{Float64}:
                                  0 1 0
0.0 0.0 0.0
                                  3 5 0
3.0 5.0 7.0
                                  4
                                     0 2
```

Matrix multiplication

Operators on scalars and matrices:

```
julia> [1 1; 1 0] * [2 3]'
2×1 Matrix{Int64}:
5
2
julia> [1 2 3] .* [10 10 15]
1×3 Matrix{Int64}:
10 20 45
```

```
Inner and outer vector product:
  julia> [2 3 5] * [1 7 11]'
  1×1 Matrix{Int64}:
    78
  julia> [2 3 5]' * [1 7 11]
  3×3 Matrix{Int64}:
    2 14 22
    3 21 33
```

5 35 55

Complex number types: Complex{Int16}, Complex{Float64}, etc.

The imaginary unit vector $\sqrt{-1}$ is available as 1im and and vectors and matrices can also be complex.

Related functions: real, imag, conj, exp, cis, abs, angle

Plotting

using Plots



Function plot expects a vectors of x coordinates, and a vector or matrix of y coordinates, one column per curve. Use plot! to add additional curves with independent x coordinates.

```
Use savefig("plot2.pdf") to save current figure as graphics file.
```

2D plotting

using Plots



Plots.jl manual: https://docs.juliaplots.org/

Functions

```
To define a new function, for example decibel(x) = 10^{x/20}, write
function decibel(x)
return 10 .^ (x ./ 20)
end
or simply
decibel(x) = 10 .^ (x ./ 20)
and call as
julia> decibel(40)
```

```
100.0
```

Note that the function needs no type declaration for parameters. Each time the function is called with a new type, a new *method* will be JIT compiled for that type signature.

Type annotations using :: are assertions and for type-dependent dispatch.

Default values for positional and keyword parameters:

```
function decibel(x=0; base::Number=10)
return base .^ (x ./ 20)
end
```

Example: generating an audio illusion

Generate an audio file with 12 sine tones of apparently continuously exponentially increasing frequency, which never leave the frequency range 300–3400 Hz. Do this by letting them wrap around the frequency interval and reduce their volume near the interval boundaries based on a raised-cosine curve applied to the logarithm of the frequency.

First produce a 2 s long waveform in which each tone raises 1/12 of the frequency range, then concatenate that into a 60 s long 16-bit WAV file, mono, with 16 kHz sampling rate. Avoid phase jumps.

Parameters:

fs	=	16000;	#	sampling frequency [Hz]	
d	=	2;	#	time after which waveform repeats [s	3]
n	=	12;	#	number of tones	
fmin	=	300;	#	lowest frequency	
fmax	=	3400;	#	highest frequency	

A variant of this audio effect, where each tone is exactly one octave (factor 2 in frequency) from the next, is known as the *Shepard–Risset glissando*.

What changes to the parameters would produce that?

Spectrogram of the first 6 s:



```
Example solution:
using DSP, Plots, WAV
t = 0:1/fs:d-1/fs; # timestamps for each sample point
# normalized logarithm of frequency of each tone (row)
# for each sample point (column), all rising linearly
# from 0 to 1, then wrap around back to 0
l = mod.(((0:n-1)/n) + (t/(d*n))', 1);
f = fmin * (fmax/fmin) .^ l; # freq. for each tone and sample
p = 2*pi * cumsum(f, dims=2) / fs; # phase for each tone and sample
# make last column a multiple of 2*pi for phase continuity
p = ((2*pi*floor.(p[:,end]/(2*pi))) ./ p[:,end]) .* p;
s = sin.(p); # sine value for each tone and sample
# mixing amplitudes from raised-cosine curve over frequency
a = 0.5 .- 0.5 * cos.(2*pi * 1);
```

```
w = sum(s .* a, dims=1)/n; # mix tones together, normalize to [-1, +1]
```

Running Julia code

There are many ways to run Julia code:

Load script manually from REPL: julia> include("script.jl")

Run as script:

\$ julia script.jl

Run as script, then activate REPL (e.g., to manually call functions):

\$ julia -i script.jl

- Automatically reload modified source files: https://github.com/timholy/Revise.jl
- Run from within an IDE, such as Visual Studio Code: https://code.visualstudio.com/docs/languages/julia
- Jupyter notebooks for Julia, Python, R, etc. https://github.com/JuliaLang/IJulia.jl
- Pluto.jl notebooks reactive notebooks that are Julia scripts https://github.com/fonsp/Pluto.jl

Pluto notebooks

- ▶ Web browser + JavaScript based working environment, Julia server
- ▶ Notebook is a sequence of "cells", each with a Julia expression
- ► Add a new cell by clicking "+" above/below existing cell
- Run a cell by pressing Shift+Enter
- Return value of the expression in a cell is displayed above the cell
- Notebooks can be exported as PDF or static HTML
- Cell can output pretty documentation as Markdown md"..." or HTML html"..." strings, cell source code can be hidden

Pluto notebooks are "reactive", like a spreadsheet

- Each global variable can only be assigned to in one cell
- If running that cell changes a global variable, then all other cells that read that global variable get automatically re-evaluated
- A Pluto notebook is just a Julia script containing the code from all cell, arranged in the order in which they need to be executed

The order in which cells appear in the browser does not matter To run multiple expressions in the same Pluto cell, wrap them in a begin ... end block. To create a lexical scope for local variables in a cell, use instead let ... end blocks or functions.

Cheatsheet: finding out things in Julia

typeof(...) sizeof(...) axes(...) eachindex(...) eltype(...) apropos("keyword") methods(open) methodswith(Vector{UInt8}) Oshow expr $dump(\ldots)$ fieldnames(Complex) Complex{Real}.types names(Base) subtypes(AbstractString) supertype(String)

type of any object array dimensions array index ranges vector index range element type of an Array search in documentation for string list all methods for a function list all methods that accept a type show expression and result, return result field types and values of structs fieldnames of a struct types of struct fields names exported by a module list of immediate subtypes of a type return the supertype of a type

@code_lowered, @code_typed, @code_warntype, @code_llvm, @code_native show code in different compilation stages, @edit, @less give easy access to source code.

MATLAB, Julia, NumPy: comparative cheat sheet

	MATLAB	Julia	NumPy
vector size (1,n)	[1 2 3]	[1 2 3]	np.array([1, 2, 3]).reshape(1, 3)
vector size (n,1)	[1; 2; 3]	[1 2 3]'	np.array([1, 2, 3]).reshape(3, 1)
vector size (n)	n/a	[1, 2, 3]	np.array([1, 2, 3])
j to n step k	j:k:n	j:k:n	np.arange(j, n+1, k)
matrix	[1 2; 3 4]	[1 2; 3 4]	np.array([[1, 2], [3, 4]])
0 matrix 1 matrix identity matrix	zeros(2, 2) ones(2, 2)	zeros(2, 2) ones(2, 2) I	np.zeros((2, 2)) np.ones((2, 2)) np.eve(2)
diagonal matrix	diag([1 2 3])	Diagonal([1, 2, 3])	np.diag([1, 2, 3])
transpose	A.'	transpose(A)	A.T
complex conj. transpose concat hor.	A' [[1 2] [1 2]]	A' [[1 2] [1 2]]	A.conj() B = np.array([1, 2]) np.hstack((B, B))
matrix to vector flip left/right broadcast a function	A(:) fliplr(A) f=@(x) x.^2; f(x)	A[:] reverse(A,dims=2) f(x)=x^2; f.(x)	A.flatten() np.fliplr(A) def f(x): return x**2 f(x)
element $A_{2,2}$	A(2, 2)	A[2, 2]	A[1, 1]
rows 1 to 4	A(1:4, :)	A[1:4, :]	A[0:4, :]
element-wise multipl.	A .* B	A .* B	A * B
matrix multiplication	A * B	A * B	A @ B

https://cheatsheets.quantecon.org/