

Julia - an overview

Jürgen Fuhrmann
WIAS Berlin



☰ Table of Contents

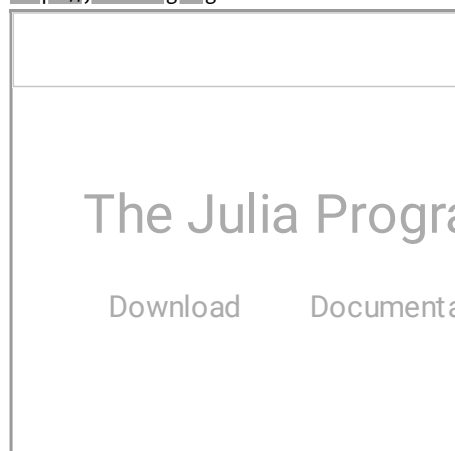
Julia: A fresh approach to numerical computing
How to install and run Julia
Pluto notebooks
Package management
Julia packages
Binary packages
Julia type system
Functions, Methods and Multiple Dispatch
Automatic differentiation (AD)
A rudimentary dual number example
Dual numbers in Julia: ForwardDiff.jl
A simple Newton solver
DifferentialEquations.jl
Some Julia Caveats
Summary

Julia: A fresh approach to numerical computing

S. Bezanson, A. Edelman, S. Karpinski, V. Shah, SIAM Review Vol. 59 (2017), No. 1, pp. 65–98

- Use of modern knowledge in language design
- Syntax comparable to matlab, python/numpy
- Just-ahead-of time compilation to native code
⇒ high performance without need for vectorization
- Built-in performant multi-dimensional arrays, comprehensive linear algebra
- Efficient integration with C/C++, python, R...
- Parallelization: SIMD, threading, distributed memory
- Composable package ecosystem focused on data science and scientific computing
- Open source (MIT License)

<https://julialang.org>



How to install and run Julia

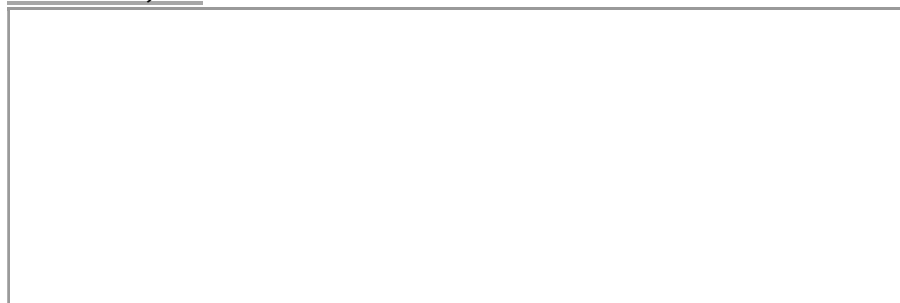
Installation:

- [Download](#) from julialang.org
 - Linux, Windows, MacOS
 - Version 1.10 with long term support
 - Current version 1.11

Modern Julia Workflows



From Zero to Julia!

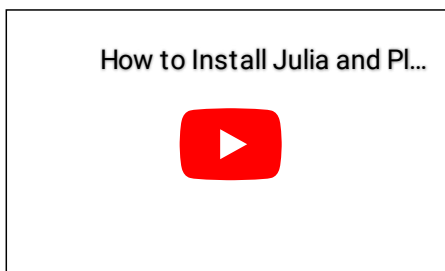


- Edit source code in any editor, run the code in the Julia REPL (command line utility)
- Visual Studio Code Julia plugin
- Jupyter notebooks in the browser
- Pluto notebooks in the browser

Pluto notebooks

Pluto is a browser based notebook interface for the Julia language. It allows to present Julia code and computational results in a tightly linked fashion.

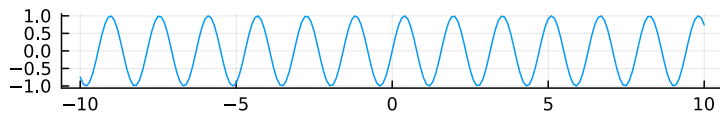
- [Pluto repository at Github](#)
- Pluto is easily installed via Julia's inbuilt package manager.
- **This presentation** is a pluto notebook.



Pluto notebooks are **reactive**: dependent cells are automatically recalculated:

```
1 using Plots
```

```
1 f(x)=sin(4x);
```



```
1 X=-10:0.1:10; plot(X,f.(X),size=(500,80),label="")
```

Package management

- **Packages** extend Julia's core functionality. Each package is a git repository with standardized structure.
- **Package Registries** provide the infrastructure for finding package repositories via package names
- The default **General Registry** has ≈ 10000 Julia packages and ≈ 1500 binary packages
- **Package Environments**
 - record version compatibility (in `Project.toml`)
 - record exact package versions (in `Manifest.toml`)
 - they are transferable and thus allow for 100% **reproducible code**
 - Add a package to the environment and use it:
`Pkg.add("ForwardDiff"); using ForwardDiff`
- Pluto notebooks include their own environment and package manager, here we can omit the `add` statement

xkcd.com

```
INSTALL.SH
#!/bin/bash

pip install "$1" &
easy_install "$1" &
brew install "$1" &
npm install "$1" &
yum install "$1" & dnf install "$1" &
docker run "$1" &
pkg install "$1" &
apt-get install "$1" &
sudo apt-get install "$1" &
steamcmd +app_update "$1" validate &
git clone https://github.com/"$1"/"$1" &
cd "$1";./configure;make;make install &
curl "$1" | bash &
```

Julia packages

- Julia packages have a standardized structure which includes the interface to the package itself, unit testing and documentation
- A package's `Project.toml` describes dependencies and version compatibility
- Some interesting packages:
 - Flux.jl, Lux.jl, ML.jl, Knet.jl, AlphaZero.jl: Machine learning
 - Jump.jl: Modeling language for Mathematical Optimization
 - Turing.jl: Bayesian inference with probabilistic programming.
 - GLMakie.jl: Plotting with GPU support, CairoMakie.jl
 - PyPlot.jl: Interface to python/matplotlib
 - ForwardDiff.jl, Enzyme.jl, Zygote.jl: Automatic differentiation
 - ModelingToolkit.jl: Modeling framework for automatically parallelized scientific machine learning (SciML) in Julia. A computer algebra system for integrated symbolics for physics-informed machine learning and automated transformations of differential equations
 - Distributions.jl: Probability distributions and associated functions.
 - CUDA.jl, CUDArrays.jl: Julia on GPU
 - Gridap.jl: Finite element package
 - BifurcationKit.jl: Bifurcation analysis
 - Franklin.jl: Static website builder

Binary packages

- The **Foreign Function Interface (FFI)** allows to call binary code compliant with the C ABI
- `CxxWrap.jl`, `CXX.jl` provide ways to interoperate with code compiled from C++
- `BinaryBuilder.jl` provides a container like infrastructure to build and maintain **binary packages** for **all architectures supported by Julia**

- Leverage of software libraries from the world of compiled languages
- Access to sparse solvers, preconditioners, mesh generators, GUI toolkits etc.
- Binary packages are handled by the package manager as well
- Binary packages contain compiled code for all operating systems supported by Julia

```
1 using GSL_jll
```

```
0.15064525725099692
```

```
1 ccall((:gsl_sf_bessel_J0, GSL\_jll.libgsl), Cdouble, (Cdouble,), 6.0)
```

Usually, wrapper packages with Julia like API are created around binary packages:

```
1 using GSL
```

```
0.15064525725099692
```

```
1 GSL.sf\_bessel\_J0(6.0)
```

Julia type system

- Julia is a strongly typed language
- Knowledge about the layout of a value in memory is encoded in its type and prerequisite for compiling to performant machine code
- Concrete types
 - Every value in Julia has a concrete type
 - Concrete types correspond to computer representations of objects
 - Built-in types deduced from concrete representations
 - User defined types via structs

```
a = 1.0
```

```
1 a=1.0
```

```
Float64
```

```
1 typeof(a)
```

- Abstract types label concepts which work for a several concrete types without regard to their memory layout etc

Functions, Methods and Multiple Dispatch

Julia Functions can have different variants of their implementation depending on the types of parameters passed to them.

- These variants are called **methods**
- The act of figuring out which method of a function to call depending on the type of parameters is called **multiple dispatch**

Define a function `test_dispatch`:

```
test_dispatch (generic function with 4 methods)
```

```
1 begin
2     test_dispatch(x)="general case: \$\(typeof\(x\)\), x=\$\(x\)"
3     test_dispatch(x::Float64)="special case Float64, x=\$\(x\)"
4     test_dispatch(x::Int64)="special case Int64, x=\$\(x\)"
5     test_dispatch(x::AbstractArray)="abstract array: \$\(typeof\(x\)\),
   size=\$\(size\(x\)\)"
6 end
```

```
"general case: String, x=xxx"
```

```
1 test_dispatch("xxx")
```

4 methods for generic function **test_dispatch** from Main.var"workspace#9":

- test_dispatch(x::Int64) in Main.var"workspace#9" at </home/fuhrmann/Wias/events/2024/2024-11-saclay/VoronoiFVOT/JuliaOverview.jl#0319ee9b-3670-4c4c-9aa8-eabb7b3dfae6:4>
- test_dispatch(x::Float64) in Main.var"workspace#9" at </home/fuhrmann/Wias/events/2024/2024-11-saclay/VoronoiFVOT/JuliaOverview.jl#0319ee9b-3670-4c4c-9aa8-eabb7b3dfae6:3>
- test_dispatch(x::AbstractArray) in Main.var"workspace#9" at </home/fuhrmann/Wias/events/2024/2024-11-saclay/VoronoiFVOT/JuliaOverview.jl#0319ee9b-3670-4c4c-9aa8-eabb7b3dfae6:5>
- test_dispatch(x) in Main.var"workspace#9" at </home/fuhrmann/Wias/events/2024/2024-11-saclay/VoronoiFVOT/JuliaOverview.jl#0319ee9b-3670-4c4c-9aa8-eabb7b3dfae6:2>

```
1 methods(test_dispatch)
```

Julia generates specialized code for function methods depending on the type of function parameters **if the types of the parameters is fixed**

- Paradigm for **code structuring and code extension**
- Users can add new methods to given functions for their own datatype
- Knowledge of parameter types is a prerequisite for optimization
- **Performant code** for functions tailored to the particular parameter types

The `@code_native` macro shows the generated assembler code

```
1 square(x)=x*x;
```

```
1 @code_native square(Float32(3.0))
```

```
.text
.file "square"
.globl julia_square_3025          # -- Begin function julia_square_3025
.p2align 4, 0x90
.type julia_square_3025,@function
julia_square_3025:                # @julia_square_3025
; @ /home/fuhrmann/Wias/events/2024/2024-11-saclay/VoronoiFVOT/JuliaOverview.jl#0319ee9b-3670-4c4c-9aa8-eabb7b3dfae6:4 within 'square'
# 3bb 0:                                # stop
push rbp
mov rbp, rsp
; @ Float32{3.0} within 'a'
vmulss xmm0, xmm0, xmm0
pop rbp
ret
.Lfunc_end0:
.size julia_square_3025, .Lfunc_end0-julia_square_3025
; -- End Function
.section ".note.GNU-stack","",@progbits
```

► [MethodInstance for Main.var"workspace#9".square(::Float32)]

```
1 MethodAnalysis.methodinstances(square)
```

Automatic differentiation (AD)

Multiple dispatch can be leveraged to implement forward mode automatic differentiation

Complex numbers \mathbb{C} extend the real numbers \mathbb{R} based on the introduction of i with $i^2 = -1$.

Dual numbers \mathbb{D} extend the real numbers \mathbb{R} by introducing a number ϵ with $\epsilon^2 = 0$:

$$\mathbb{D} = \{a + b\epsilon \mid a, b \in \mathbb{R}\} = \left\{ \begin{pmatrix} a & b \\ 0 & a \end{pmatrix} \mid a, b \in \mathbb{R} \right\} \subset \mathbb{R}^{2 \times 2}$$

Polynomials of dual numbers: Let $p(x) = \sum_{i=0}^n p_i x^i$. Then

$$p(a + b\varepsilon) = \sum_{i=0}^n p_i a^i + \sum_{i=1}^n i p_i a^{i-1} b \varepsilon = p(a) + b p'(a) \varepsilon$$

- Automatic evaluation of function and derivative at once
- \Rightarrow **forward mode automatic differentiation**
- Multivariate dual numbers: generalization for **partial derivatives**

A rudimentary dual number example

Define a dual number type as a parametric type

```
1 begin
2   struct MyDual{T} <: Number where {T <: Real}
3       v::T
4       d::T
5   end
6   MyDual(x)=MyDual(x,one(x))
7 end;
```

Define some arithmetic operations for this type

```
1 begin
2   Base.:+(x::MyDual, y::MyDual) = MyDual(x.v+y.v, x.d+y.d)
3   Base.:-(y::MyDual)             = MyDual(-y.v, -y.d)
4   Base.:-(x::MyDual, y::MyDual) = x + -y
5   Base.:*(x::MyDual, y::MyDual) = MyDual(x.v*y.v, x.v*y.d + x.d*y.v)
6 end
```

```
e = ▶ MyDual(3, 1)
```

```
1 e=MyDual(3,1)
```

```
MyDual{Int64}
```

```
1 typeof(e)
```

```
1 @code_native square(e)
```

```
.text
.file "square"
.globl julia_square_3244 # -- Begin function julia_square_3244
.p2align 4, 0x90
.type julia_square_3244,@function
julia_square_3244: # @julia_square_3244
; @ /home/fuhrmann/wias/events/2024/2024-11-saclay/VoronoiFVOT/JuliaOverv
ew.jl#==#ef6e81b2-cb7b-4687-96b8-2840f8cb789a:1 within 'square'
# $bb.0: # Stop
push rbp
; @ /home/fuhrmann/wias/events/2024/2024-11-saclay/VoronoiFVOT/JuliaOverv
ew.jl#==#3c133925-c035-4351-a5cf-d0f7accff119:5 within 'a' @ int.jl:88
mov rcx, qword ptr [rsi]
mov rbp, rsp
mov rax, rdi
mov rdx, rcx
imul rdx, rcx
; @ /home/fuhrmann/wias/events/2024/2024-11-saclay/VoronoiFVOT/JuliaOverv
ew.jl#==#3c133925-c035-4351-a5cf-d0f7accff119:5 within 'a'
; @ int.jl:87 within 'a'
imul rcx, qword ptr [rsi + 8]
; @ /home/fuhrmann/wias/events/2024/2024-11-saclay/VoronoiFVOT/JuliaOverv
ew.jl#==#501b742-bd42-45a6-bd03-33a77260185f:3 within 'MyDual'
mov qword ptr [rdi], rdx
; @ int.jl:87 within 'a'
add rcx, rcx
; @ /home/fuhrmann/wias/events/2024/2024-11-saclay/VoronoiFVOT/JuliaOverv
ew.jl#==#501b742-bd42-45a6-bd03-33a77260185f:3 within 'MyDual'
mov qword ptr [rdi + 8], rcx
pop rbp
ret
.Lfunc_end0:
.size julia_square_3244, .Lfunc_end0-julia_square_3244
; -- End function
.section ".note.GNU-stack","",@progbits
```

Check with a polynomial

```
1 p(x)=x^3+x^2;
```

```
1 dp(x)=3x^2+2x;
```

```
x = 10.5
```

```
1 x=10.5
```

```
► MyDual(1267.875, 351.75)
```

```
1 p(MyDual(x))
```

Note that we did not define $^$ for dual numbers - this comes through the abstract number interface.

```
► (1267.88, 351.75)
```

```
1 p(x), dp(x)
```

Dual numbers in Julia: ForwardDiff.jl

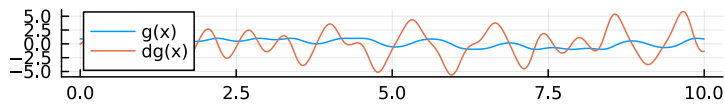
```
1 using ForwardDiff
```

```
Dual{Nothing}(1267.875, 351.75)
```

```
1 p(ForwardDiff.Dual(x,1))
```

```
1 g(x)=sin(exp(0.2*x))+cos(5x));
```

```
1 dg(x)=ForwardDiff.derivative(g,x);
```



```

1 let
2     X=0:0.01:10
3     p=plot(X,g.(X),size=(500,75),label="g(x)")
4     plot!(p,X,dg.(X),label="dg(x)")
5 end

```

A simple Newton solver

```

1 using DiffResults,LinearAlgebra

```

```

1 function newton(A,b;u0=zeros(length(b)), tol=1.0e-12, maxit=120)
2     result=DiffResults.JacobianResult(u0) # result buffer for AD
3     u=copy(u0)
4     it=1
5     while it<maxit
6         ForwardDiff.jacobian!(result,v->A(v)-b,u)
7         jac=DiffResults.jacobian(result)
8         residual=DiffResults.value(result)
9         h=jac\r residual
10        u-=h
11        δ=norm(h)
12        @info "δ=$(round(δ,digits=5))"
13        if δ<tol
14            return u
15        end
16        it=it+1
17    end
18    throw("convergence failed")
19 end;

```

Define a function A and right hand side b

```

1 A(u)=[u[1]^3+2*u[2]^5+u[1]+1, 4u[2]+1];

```

```

1 b=[0,0];

```

```

result = ▶ [-0.681512, -0.25]

```

```

1 result=newton(A,b)

```

```

① δ=1.03078

```

```

① δ=0.25049

```

```

① δ=0.06424

```

```

① δ=0.00374

```

```

① δ=1.0e-5

```

```

① δ=0.0

```

```

① δ=0.0

```

```

▶ [0.0, 0.0]

```

```

1 A(result)-b

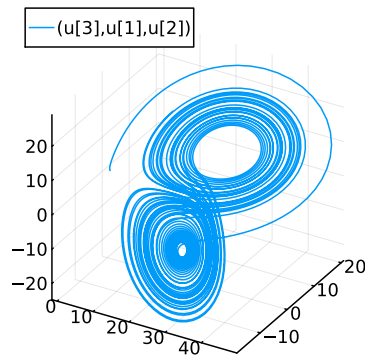
```

- Generalization to large systems of equations
- Automatic sparsity detection
- Support of sparse matrices
- NLSolve.jl package

DifferentialEquations.jl I

```
1 using DifferentialEquations
```

```
1 function lorenz!(du,u,p,t)
2   du[1] = 10.0*(u[2]-u[1])
3   du[2] = u[1]*(28.0-u[3]) - u[2]
4   du[3] = u[1]*u[2] - (8/3)*u[3]
5 end;
```




```
1 begin
2   u0 = [1.0;0.0;.0];
3   tspan = (0.0,50.0)
4   @time sol = DifferentialEquations.solve(ODEProblem(lorenz!,u0,tspan))
5   plot(sol,vars=(3,1,2),size=(300,300))
6 end
```


⚠ To maintain consistency with solution indexing, keyword argument vars will be removed in a future version. Please use keyword argument idxs instead.
caller: ip:0x0

2.967122 seconds (8.32 M allocations: 546.298 MiB, 6.99% gc time, 99.9 2% compilation time: <1% of which was recompilation) ?




- Multi-language suite for high-performance solvers of differential equations and scientific machine learning (SciML) components
- Solvers for systems of ordinary differential equations (ODEs), Differential-algebraic equations (DAEs), Stochastic DEs (SDEs), Delay DEs (DDEs), ...
- Main author: Chris Rackauckas

Just the list of DAE solver methods from https://diffeq.sciml.ai/stable/solvers/dae_solve/...


[HOME](#)
[MODELING ▾](#)
[SOLVERS ▾](#)
[ANALYSIS ▾](#)
[MACHINE LEARNING ▾](#)



[Solver Algorithms](#)
[DEVELOPER TOOLS ▾](#)

[GitHub](#)




[/ Mass Matrix and Fully Implicit DAE Solvers](#)

Mass Matrix and Fully Implicit DAE Solvers

Recommended Methods

For medium to low accuracy small numbers of DAEs in constant mass matrix form, the Rosenbrock23 and Rodas4 methods are good choices which will get good efficiency if the mass matrix is constant. Rosenbrock23 is better for low accuracy (error tolerance $<1e-4$) and Rodas4 is better for high accuracy. Another choice at high accuracy is

Some Julia Caveats

- JIT precompilation time
 - the price for generic programming and multiple dispatch
 - this has improved significantly since v1.0
- Several great plotting packages available but for covering all plotting needs (speed, portability, feature completeness) it may be necessary to use more than one
- There are some pitfalls with reaching full performance - e.g. all performant code needs to be in functions
- Interfaces are informal – something similar to C++20 concepts formalizing interface definitions would be nice to have
- Less packages than e.g. python due to relatively young age and focus on computing (instead of trying to be a general purpose language)

Summary

- Interesting new possibilities
 - Generic programming without syntax overload
 - Multiple dispatch instead of object orientation (aka single dispatch)
 - Informal interfaces supporting algorithm coupling
 - Easy access to automatic differentiation
- Ability to achieve high performance with Matlab level syntax
- Emerging package ecosystem + community focused on Scientific Computing, data analysis
- Open source
- Structural support for FAIR research paradigm

```

1 begin
2     using PlutoUI
3     import PlutoUI.ExperimentalLayout
4     using HypertextLiteral: @html, @html_str
5     EL=PlutoUI.ExperimentalLayout
6     using MethodAnalysis
7     import PlutoVista
8     using GridVisualize
9     default_plotter!(PlutoVista);
10 end;

```

```

1 hsep(px=50)=html"<div style='width:$(px)px'>"";

```

