for Physics (and Physicists)

Oliver Schulz, Ricarda Riepen

Max Planck Institute for Physics

oschulz@mpp.mpg.de, riepen@mpp.mpg.de

Advanced Programming Concepts 2025, June 2025

Why Julia?

Science needs code - but how to write it?

- Choice of programming language(s) matter!
- Need to balance:
  - Learning time
  - Productivity
  - Performance
- Usually involves compromises

Programming Language Options

- C++:
  - Pro: Very fast (in expert hands)
  - Pro: Really cool new concepts (even literally) in C++11/14/17/...
  - Con: Complex, takes long time to learn and much longer to master
  - Con: Straightforward tasks often result in lengthy code
  - Con: No memory management (General protection faults)
  - Con: No universal package management
  - Con: Composability isn't great

Programming Language Options

- Python:
  - Pro: Broad user base, popular first programming language
  - Pro: Easy to learn, good standard library
  - Con: Can't write time-critical loops in Python, workarounds like Numba/Cython have many limitations, don't compose well
  - Con: Language itself fairly primitive, not very expressive
  - Con: Duck-Typing necessitates lots of test code
  - Con: No effective multi-threading
  - Con: Composability isn't great

What else is there?

- Fortran:
  - Pro: Math can be really fast
  - Con: Old language, few modern concepts
  - Con: Shrinking user base
  - Con: Composability isn't great
  - Do you *really* want to ...?

- Scala, Go, Kotlin etc.:
  - Pro: Lots of individual strengths
  - Con: Math either fast *or* generic *or* or complicated
  - Con: Calling C, Fortran or Phython code often difficult
  - Con: Composability isn't great

The 97 and the 3 Percent

> We should forget about small efficiencies, say about 97% of the time: *premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*

Donald E. Knuth

- Some programming languages (e.g. Python) great for the 97% - but can't make the 3% fast.
- Some other languages (e.g. C/C++, Fortran) can handle the 3% - but makes the 97% complicated.

The Two-language Problem

- Common approach nowadays: Write time critical code in C/C++, rest in Python
- Pro: End-user can code comfortably in Python, with good performance
- Con: Complexity of C/C++ **plus** complexity of Python
- Con: Need proficiency in **two** languages, barrier that prevents non-expert users from contributing to important parts of code
- Con: Limits generic implementation of algorithms
- Con: Severely limits metaprogramming, automatic differentiation, etc.

The Expression Problem

> The expression problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts).

Philip Wadler

- In other words: The capability to add both new subtypes and new functionality for a type defined in a package you don't own
- Object oriented languages typically can't do this
  (Ruby has a dirty way, Scala a clean workaround)
- If you have programming experience, you have felt this, even if you didn't name it
- Result: Packages tend not to compose well

What we want is a language …

- as fast as C/C++/Fortran
- as easy to learn and productive as Python
- with a solution for the expression problem
- with first class math support (vectors, matrices, etc.)
- with excellent package management
- with true functional programming
- with great Fortran/C/C++/Python integration
- with true metaprogramming (like Lisp or Scala)
- good at parallel and distributed programming
- suitable for interactive, small and large applications

Julia

- Designed for scientific/technical computing
- Originated at MIT, first public version 2012
- Covers the whole wish-list
- Clear focus on user productivity and software quality
- Rapid growth of user base and software packages
- Current version: Julia v1.11

Julia Language Properties

- Fast: JAOT compilation to native CPU and GPU code
- Multiple-dispatch (more powerful than object-oriented):
  solves the expression problem
- Dynamically typed
- Very powerful type system, types are first-class values
- Functional programming and metaprogramming
- First-class math support (like Fortran or Matlab)
- ...

Julia Language Properties, cont.

- ...
- Local and distributed code execution
- State-of-the-art multi-threading: parallel code
  can call parallel code that can call parallel code, ...,
  without oversubscribing threads
- Software package management:
  Trivial to create and install packages
  built-in good scientific practice
- Excellent REPL (console)
- Easy to call Fortran, C/C++ and Python code

Particle physics publications with Julia

Incomplete selection of some particle physics papers partially or fully based on Julia:

- GERDA *Final Results of GERDA on the Search for Neutrinoless Double-β Decay* Phys. Rev. Lett. (2020)
- LHCb *Study of the doubly charmed tetraquark* T_{cc}^{+} nature comm., 2021
- LHCb *Observation of excited* Ω_c^0 *baryons in* $\Omega_b^- \to \Xi_c^+ K^- m\pi^-$ *decays* Phys. Rev. D, (2021)

- Eschle et al., *Potential of the Julia programming language for high energy physics computing* Comput Softw Big Sci (2023)
- Botje et al. *Constraints on the Up-Quark Valence Distribution in the Proton* PhysRevLett (2023)
- An et al., *The determination of the spin and parity of a vector-vector system* acc. by JHEP (2024)

Julia in high-performance computing (HPC)

Julia at scale (very incomplete list):

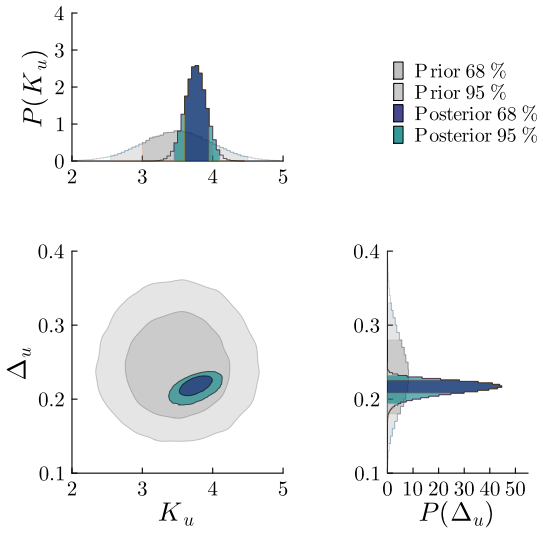
- Celeste: Variational Bayesian inference for astronomical images (doi:10.1214/19-AOAS1258), 1.54 petaflops using 1.3 million threads on 9,300 Knights Landing (KNL) nodes on Cori at NERSC
- Clima: Full-earth climate simulation, https://clima.caltech.edu, large team, uses everything from MPI to GPUs
- Ice-flow simulations with FastIce.jl on supercomputer LUMI (AMD GPUs), excellent scaling
- ...

When (not) to use Julia

- *Do* use Julia for computations, visualization, data processing ... pretty much anything scientific/technical
- *Do not* use Julia for scripts what will only run for a second (code gen overhead), use Python or shell scripts
- *Do not (at least not yet)* use Julia for (non-computational) web apps, etc., use Go or Node.js
- *Do not (at least not yet)* use Julia for big machine learning with standard building blocks (LLMs, etc.), use Python frameworks
- *Do* try Julia for custom machine learning that mixes physics models and ML blocks

Use case: ZEUS ep-collision parton PDF fit

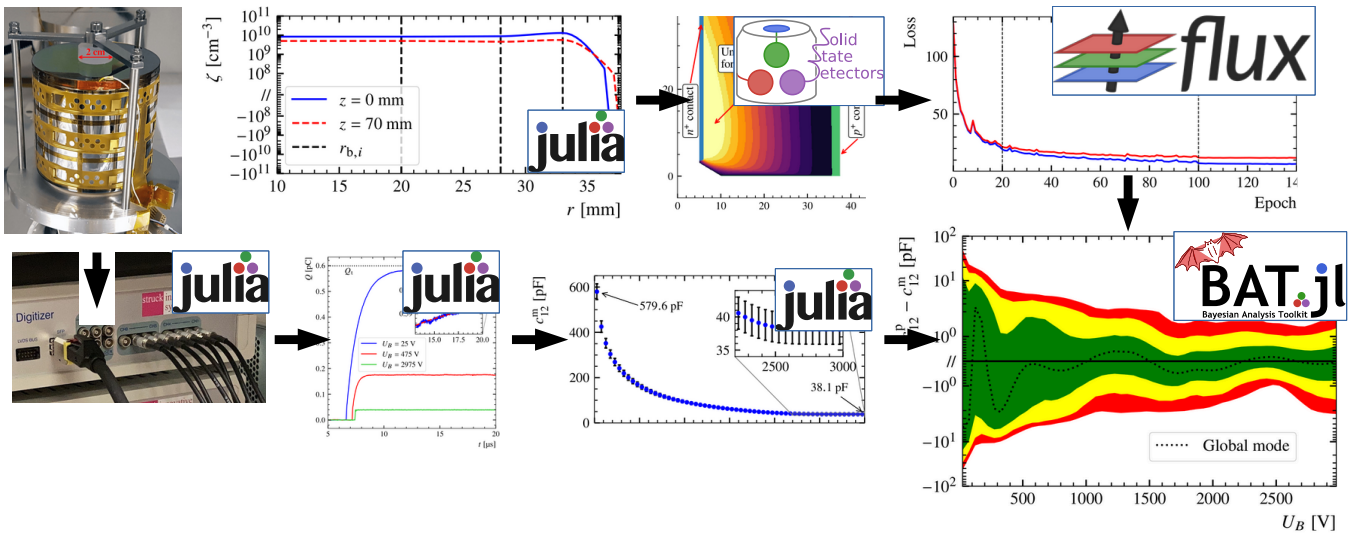
QCDNUM (Fortran) wrapped in Julia, samples with BAT.jl



[PRL.130.141901]

Use case: Determining HPGe detector impurities

Bayesian inference of longitudinal and radial impurity profiles from CV-measurements:



[EPJ-C 2023, https://doi.org/10.1140/epjc/s10052-023-11509-8]

The Julia language

Verbs and nouns - functions and types

- Julia is not Java: Verbs aren't owned by nouns
- Julia has: types, functions and methods
- Object-oriented languages: methods belong to types
- Julia: methods belong to *functions*, not to types!
  In combination with multiple dispatch, much more powerful than OO.

One-liner functions

Short one-liner function:

```
f1 (generic function with 1 method)
```

```
f1(x) = x^2
```

```
9
```

```
f1(3)
```

Function that needs more than one line:

Multi-line funcions

```
function f2(x)
    # ... something ...
    x^2
end
```

is equivalent to

```
function f2(x)
    # ... something ...
    return x^2
end
```

The result of the last expression is returned by default
(like in Mathematica)

```
f4 (generic function with 1 method)
```

Note: `return` is optional, and often not used explicitly. Last expression in a function, block, etc. is automatically returned (like in Mathematica).

Types

An abstract type, must be empty:

```
abstract type MySuperType end
```

An immutable type, value of `i` can't change:

```
struct MySubType <: MySuperType
    i::Int
    j::Int
end
```

A mutable type, value of `i` can change:

```
mutable struct MyMutableSubType <: MySuperType
    i::Int
    j::Int
end
```

Parametric types

Julia has a powerful parametric type system, based on set theory:

```
f(A::AbstractMatrix{<:Real}) = do_something_with(A)
```

implements function `f` for *any* real-valued matrix,
but we may need to handle complex numbers differently:

```
f(A::AbstractMatrix{<:Complex}) = do_something_else_with(A)
```

Julia makes this easy!

Semantics: `AbstractMatrix{<:Real} == AbstractMatrix{T} where {T<:Real}` is the set of all matrix types that have the element type parameter set to an element of the set of all real number types.

Type aliases and union types

Type aliases are basically just const values:

```
const Abstract2DArray{T} = AbstractArray{T,2}
rand(2, 2) isa Abstract2DArray == true
```

Type unions are unions of set of types.

```
const RealVecOrMat{T} where {T<:Real} = Union{AbstractArray{T,1}, AbstractArray{T,
2}}
```

is the union of a 1D and 2D array types with real-valued elements.

Syntax: Variables

```
# Global variables:
const a = 42
b = 24

function foo(x)
    # Local variables:

    c = a * x
    d = b * x # Avoid, type of b can change!
    #...
end
```

Loops

For loop:

```
for i in something_iterable
    # ...
end
```

`something_iterable` can be a range, an array, anything that implements the Julia iterator API.

While loop:

```
while condition
    # do something
end
```

Control flow

If-else, evaluate only one branch:

```
if condition
    # do something
elseif condition
    # do something else
else
    # or something different
end
```

Ternary operator, evaluate only one branch:

```
condition ? result_if_true : result_if_false
```

`ifelse`, evaluate both results but return only one:

```
ifelse(condition, result_if_true, result_if_false)
```

All of these can return a value!

Blocks and scoping

Begin/end-block (does *not* introduce a new scope):

```
begin
    # *Not* a new scope in here
    # ...
end
```

Let-block (*does* introduce a new scope):

```
b = 24

let my_b = b
    # New scope in here.
    # If b is bound to new value, my_b won't change.
    # ...
end
```

Arrays

Vectors:

```
v = [1, 2, 3]

v = rand(5)
```

Matrices:

```
A = [1 2; 3 4]

A = rand(4, 5)
```

- Column-first memory layout!
- Almost anything array-like is subtype of `AbstractArray`.

Array indexing

Get `i`-th element of vector `v`:

```
v[i]
```

Most higher-dimensional array types support cartesian and linear indexing (usually faster):

```
A[i, j]
A[lin_idx]
```

Use `eachindex(A)` to get indices of best type for given `A` (usually linear).

In Julia, anything array-like can usually be an index as well

```
A[2:3, [1, 4, 5]]
```

Array comprehension and generators

Returns an array:

```
[f(x) for x in some_collection]
```

Returns an iterable generator:

```
(f(x) for x in some_collection)
```

Hello World (and more) in Julia

```
1 println("Hello, World!")
```

```
Hello, World!
```

Let's define a function

```
f (generic function with 1 method)
```

```
1 f(x, y) = x * y
```

```
42.0
```

```
1 f(20, 2.1)
```

Multiplication is also defined for vectors, so this works, too:

```
[4.2, 8.4, 12.6, 16.8]
```

```
1 f(4.2, [1, 2, 3, 4])
```

Multiple Dispatch

```
foo (generic function with 1 method)
```

```
1 foo(x::Integer, y::Number) = x * y
```

```
foo (generic function with 2 methods)
```

```
1 foo(x::Integer, y::AbstractString) = join(fill(y, x))
```

```
12
```

```
1 foo(3, 4)
```

```
"abcabcabc"
```

```
1 foo(3, "abc")
```

```
1 try
2     foo(4.5, 3)
3 catch err
4     @error err
5 end
```

```
MethodError(foo (generic function with 2 methods), (4.5, 3), 0x0000000000006aca)
```

Functional Programming

```
myarray =
 [0.569897, 0.215733, 0.348129, 0.865132, 0.395845, 0.606517, 0.93473, 0.351726, 0.243129, 0.
```

```
1 myarray = rand(10)
```

```
idxs =  [1, 2, 3, 5, 8, 9, 10]
```

```
1 idxs = findall(x -> 0.2 < x < 0.6, myarray)
```

```
 [0.569897, 0.215733, 0.348129, 0.395845, 0.351726, 0.243129, 0.342446]
```

```
1 myarray[idxs]
```

Even types are first-class values:

```
 [AbstractFloat, AbstractIrrational, FixedPoint, Dual, Percentile, Integer, Logarithmic, ULo
```

```
1 subtypes(Real)
```

Julia type hierarchy extends all the way down to primitive types:

```
true
```

```
1 Float64 <: AbstractFloat <: Real <: Number <: Any
```

Broadcasting

```
1 A, B = [1.1, 2.2, 3.3], [4.4, 5.5, 6.6];
```

```
[30.25, 59.29, 98.01]
```

```
1 broadcast((x, y) -> (x + y)^2, A, B)
```

Shorter broadcast syntax:

```
[30.25, 59.29, 98.01]
```

```
1 (A .+ B) .^ 2
```

Julia compiler flow

1. Julia Code (`@less`, `@which`, `@edit`)
2. Julia AST (`@code_lowered`)
3. Julia typed IR (`@code_typed`)
4. LLVM IR (`@code_llvm`)
5. Native assembly code (`@code_native`)
6. Binary machine code

Let's Look Under the Hood

```
1 @code_llvm debuginfo=:none f(20, 2.1)
```

```
; Function Signature: f(Int64, Float64)
define double @julia_f_19417(i64 signext %"x::Int64", double %"y::Float64") #0 {
top:
  %0 = sitofp i64 %"x::Int64" to double
  %1 = fmul double %0, %"y::Float64"
  ret double %1
}
```

```
1 @code_native debuginfo=:none f(20, 2.1)
```

```
    .text
    .file   "f"
    .globl  julia_f_19609                   # -- Begin function julia_f_19609
    .p2align    4, 0x90
    .type   julia_f_19609,@function
julia_f_19609:                              # @julia_f_19609
; Function Signature: f(Int64, Float64)
# %bb.0:                                    # %top
    #DEBUG_VALUE: f:x <- $rdi
    #DEBUG_VALUE: f:y <- $xmm0
    push    rbp
    vcvtsi2sd   xmm1, xmm1, rdi
    mov rbp, rsp
    vmulsd  xmm0, xmm1, xmm0
    pop rbp
    ret
.Lfunc_end0:
    .size   julia_f_19609, .Lfunc_end0-julia_f_19609
                                            # -- End function
    .type   ".L+Core.Float64#19611",@object # @"+Core.Float64#19611"
    .section    .rodata,"a",@progbits
    .p2align    3, 0x0
".L+Core.Float64#19611":
    .quad   ".L+Core.Float64#19611.jit"
    .size   ".L+Core.Float64#19611", 8

.set ".L+Core.Float64#19611.jit", 125888626047120
    .size   ".L+Core.Float64#19611.jit", 8
    .section    ".note.GNU-stack","",@progbits
```

Loop Fusion and SIMD Vectorization

```
begin
    bcsqadd(X, Y) = (X .+ Y) .^ 2
    @code_llvm raw=false debuginfo=:none bcsqadd(A, B)
end
```

```
; Function Signature: bcsqadd(Array{Float64, 1}, Array{Float64, 1})
define nonnull ptr @julia_bcsqadd_19704(ptr noundef nonnull align 8 derefere
nceable(24) %"X::Array", ptr noundef nonnull align 8 dereferenceable(24)
%"Y::Array") #0 {
top:
  %gcframe1 = alloca [5 x ptr], align 16
  call void @llvm.memset.p0.i64(ptr align 16 %gcframe1, i8 0, i64 40, i1 tru
e)
  %thread_ptr = call ptr asm "movq %fs:0, $0", "=r"() #15
  %tls_ppgcstack = getelementptr i8, ptr %thread_ptr, i64 -8
  %tls_pgcstack = load ptr, ptr %tls_ppgcstack, align 8
  store i64 12, ptr %gcframe1, align 16
  %frame.prev = getelementptr inbounds ptr, ptr %gcframe1, i64 1
  %task.gcstack = load ptr, ptr %tls_pgcstack, align 8
  store ptr %task.gcstack, ptr %frame.prev, align 8
  store ptr %gcframe1, ptr %tls_pgcstack, align 8
  %0 = getelementptr inbounds i8, ptr %"X::Array", i64 16
  %1 = load i64, ptr %0, align 8
  %2 = getelementptr inbounds i8, ptr %"Y::Array", i64 16
  %3 = load i64, ptr %2, align 8
  %4 = icmp ne i64 %3, %1
  %value_phi.v377 = icmp ne i64 %1, 1
  %value_phi.v.not = and i1 %4, %value_phi.v377
  br i1 %value_phi.v.not, label %L20, label %L33

L20:                                              ; preds = %top
  %value_phi264.v405.not = icmp eq i64 %3, 1
  br i1 %value_phi264.v405.not, label %L33, label %L28

L28:                                              ; preds = %L20
  %ptls_field = getelementptr inbounds ptr, ptr %tls_pgcstack, i64 2
  %ptls_load = load ptr, ptr %ptls_field, align 8
  %"new::LazyString" = call noalias nonnull align 8 dereferenceable(32) ptr
@jl_gc_pool_alloc_instrumented(ptr %ptls_load, i32 848, i32 32, i64 1258885
```

Native SIMD code

```
@code_native debuginfo=:none bcsqadd(A, B)
```

```
    .text
    .file   "bcsqadd"
    .globl  julia_bcsqadd_19778             # -- Begin function julia_bcsqadd_19778
    .p2align    4, 0x90
    .type   julia_bcsqadd_19778,@function
julia_bcsqadd_19778:                    # @julia_bcsqadd_19778
; Function Signature: bcsqadd(Array{Float64, 1}, Array{Float64, 1})
# %bb.0:                                # %top
    #DEBUG_VALUE: bcsqadd:X <- [DW_OP_deref] $rdi
    #DEBUG_VALUE: bcsqadd:Y <- [DW_OP_deref] $rsi
    push    rbp
    mov rbp, rsp
    push    r15
    push    r14
    push    r13
    push    r12
    push    rbx
    sub rsp, 88
    vxorpd  xmm0, xmm0, xmm0
    #APP
    mov rax, qword ptr fs:[0]
    #NO_APP
    lea rdx, [rbp - 112]
    vmovupd ymmword ptr [rbp - 112], ymm0
    mov qword ptr [rbp - 80], 0
    mov qword ptr [rbp - 48], rdi       # 8-byte Spill
    mov rcx, qword ptr [rax - 8]
    mov qword ptr [rbp - 112], 12
    mov rax, qword ptr [rcx]
    mov qword ptr [rbp - 72], rcx       # 8-byte Spill
    mov qword ptr [rbp - 104], rax
    mov qword ptr [rcx], rdx
    #DEBUG_VALUE: bcsqadd:X <- [DW_OP_deref] 0
```

Types as first-class values

This is efficient (not runtime reflection):

```
md"""
## Types as first-class values

This is efficient (not runtime reflection):"""
```

```
32767.5
```

```
begin
    half_dynrange(T::Type{<:Number}) = (Int(typemax(T)) - Int(typemin(T))) / 2
    half_dynrange(Int16)
end
```

```
1 @code_llvm half_dynrange(Int16)
```

```
; Function Signature: half_dynrange(Type{Int16})
;  @ /home/oli/Data/Science/Projects/Julia/events/teaching/julia-for-physics/jul
ia_physics_slides.jl#==#d4e3f14f-dffb-45d6-b079-5aecd37cbfe6:2 within `half_dynr
ange`
define double @julia_half_dynrange_19868() #0 {
top:
  ret double 3.276750e+04
}
```

Package management

- Julia probably has the best package management to date
- Press "]" to enter package management console
- Typically `add PACKAGE_NAME` is sufficient, can also do `add PACKAGE_NAME@VERSION`
- To get an unreleased version, use `add PACKAGE_NAME#BRANCH_NAME`
- Easy to start modifying a package via `dev PACKAGE_NAME`
- Multiple package versions can be installed, selection via Pkg.jl environments.
- Also useful: `julia> using Pkg; pkg"<Pkg console command>"`

Package creation

- A Julia package needs:
  - A "Project.toml" file
  - A "src/PackageName.jl" file
- That's it: Push to GitHub, and package is installable via `add PACKAGE_URL`
- Use Documenter.jl to document your package
- To enable `add PACKAGE_NAME`, package must be registered, there are some rules
- Use PkgTemplates.jl to generate new package with CI config (Travis, Appveyor, ...), docs generation, etc.

No free lunch

- Package loading and code-gen can take some time, but mitigations available:
- Revise.jl: Hot code-reloading at runtime
- More and more packages use new Julia capabilities to precompile binary code

Performance tips

- Read the official Julia performance tips!
- Do *not* call on (non-const) global variables from time-critical code
- Type-stable code is fast code. Use `@code_warntype` and `Test.@inferred` to check!

- In some situations, closures can be troublesome, using `let` can help the compiler

SIMD

Demo

Shared-memory parallelism

- Julia has native multithreading support
- Simple cases: Use `@threads` macro
- Since Julia v1.3: Cache-efficient composable multi-threaded parallelism

Processes, Clusters, MPI

- Julia brings a full API for remote processes and compute clusters
- Native support for local processes and remote processes via SSH, SLURM, MPI, ...

Benchmarking and profiling, digging deeper

Demo

Docs and help

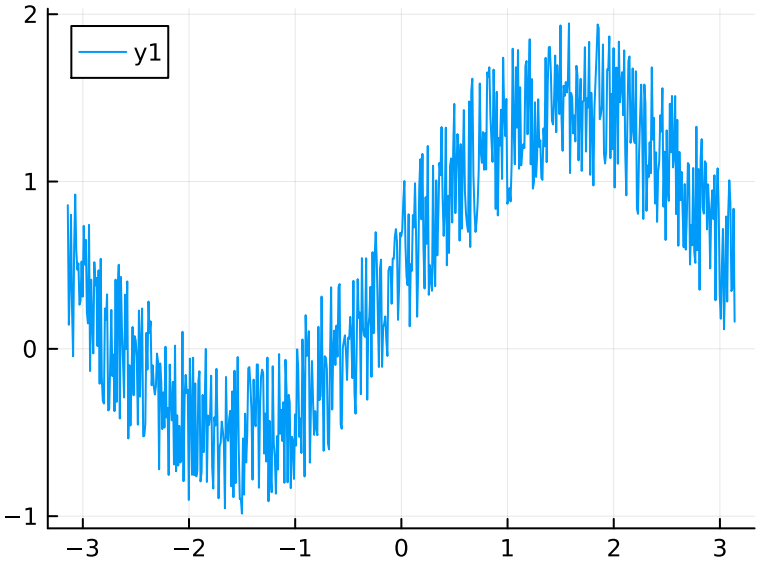
- Official Julia docs
- Julia Cheat Sheet
- Learning Julia
- Julia Discourse
- Julia Slack
- Julia on Youtube

Visualization/Plotting: Plots, Makie, plotting recipes

Let's Make a Plot

```
1 using Plots
```

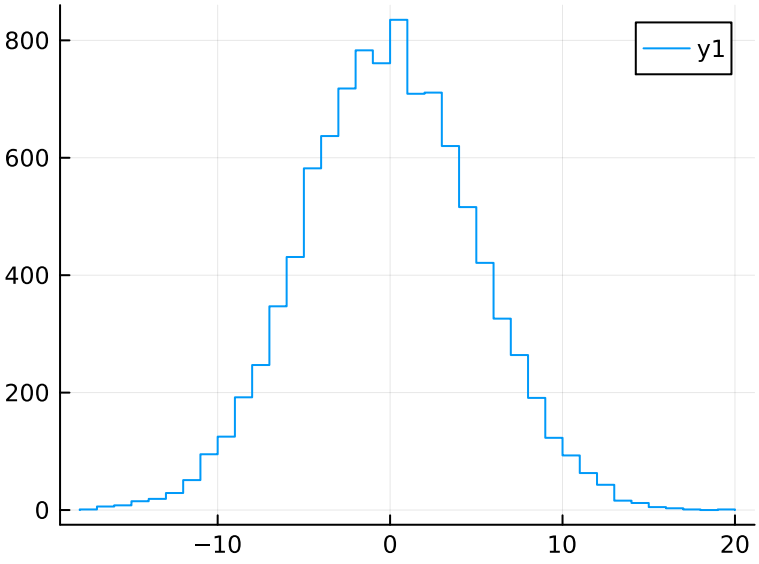
```
Plots.gr(size = (400,300));
```




```
begin
    x_range = -π:0.01:π
    plot(x_range, sin.(x_range) + rand(length(x_range)))
end
```

Histograms are easy, too

```
Distributions.Normal{Float64}(μ=0.0, σ=5.0)
```




```
stephist(rand(dist, 10^4), size = (400, 300))
```

Talking to Python

Calling Python from Julia is easy, can even use inline Python code:

```
using PyCall
numpy = pyimport("numpy")
A_jl = rand(5);
py"type($A_jl)" isa PyObject
```

Automatic differentiation

Let's define a simple neural network layer and loss function and auto-differentiate through it.

```
begin
    struct ADenseLayer{
        M<:AbstractMatrix{<:Real},V<:AbstractVector{<:Real},F<:Function
    } <: Function
        A::M
        b::V
        f::F
    end

    (l::ADenseLayer)(x::AbstractVector{<:Real}) = (l.f).(l.A * x + l.b)
end
```

Instantiating the layer

```
f_loss(y) = sum(y .^ 2);
```

```
relu(x) = ifelse(x > zero(x), x, zero(x));
```

```
mylayer = ADenseLayer(rand(5,5), rand(5), relu);
```

Evaluating and gradients

```
[2.40827, 1.19717, 2.30326, 1.97818, 2.21365]
```

```
begin
    x = rand(5)
    mylayer(x)
end
```

```
21.351496031593026
```

```
f_loss(mylayer(x))
```

```
5×5 Matrix{Float64}:
 2.96668  0.75409   4.66562  2.09928  2.99144
 1.47476  0.374864  2.31931  1.04357  1.48707
 2.83732  0.721209  4.46218  2.00774  2.86101
 2.43687  0.619418  3.83239  1.72437  2.45721
 2.72693  0.693149  4.28857  1.92963  2.74969
```

```
begin
    using Zygote
    g = Zygote.gradient((mylayer, x) -> f_loss(mylayer(x)), mylayer, x)
    g[1].A
end
```

```
[4.81655, 2.39434, 4.60653, 3.95637, 4.42731]
```

Julia sets in Julia

```
using Colors, ColorSchemes, Images
```

Julia sets are the points where iteration of $z_{i+1} = z_i^2 + c$ stays bounded.

```
f_jls(z, c) = @fastmath z^2 + c;
```

Function to count for how many iterations $|z| < 8$:

```
function n_f_jls(z, c, nmax)
    n = 0
    while abs2(z) < 8 && n < nmax
        z = f_jls(z, c)
        n += 1
    end
    return n - 1
end;
```

Increase `n_iterations` from `10^3` to `10^4` or to `10^5` to show "almost connected" sets in higher quality, for example at c = -1.2387388f0 - 0.082582586f0im . Will make things slower, enable CUDA above to speed things up significantly.

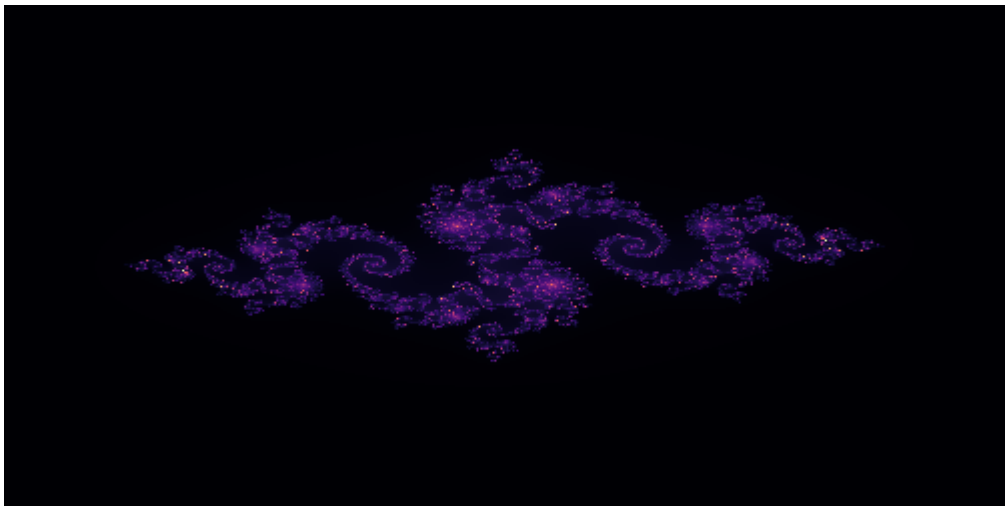
1000

```
@bind n_iterations Select([10^3, 10^4, 10^5])
```

Interactive Julia-Set plots

```
use_cuda, c_re, c_im =
```

(☐ , ,)



```
let c = T(c_re) + T(c_im) * im

    @time @. N_iter = n_f_jls(eval_points, c, n_iterations)

    N_plot = copyto!(N_plotbuf, N_iter)
    #heatmap(N_plot, ratio = 1, format = :jpg)
    @info "max iterations: $(maximum(N_iter))"
    get.(Ref(ColorSchemes.magma), N_plot .* inv(maximum(N_plot)))
end
```

```
max iterations: 533
```

```
  0.003366 seconds (2 allocations: 48 bytes)
```

```
MyArray = use_cuda ? eval(:(import CUDA; CUDA.CuArray)) : Array;
```

```
begin
    T = Float32
    # Whether we compute on GPU or GPU just depends on the array type:
    N_iter = MyArray{Int}(undef, 250, 500)
    N_plotbuf = Array{Int}(undef, size(N_iter))

    range_re = range(T(-2), T(2), length = size(N_iter, 2))
    range_im = range(T(-2), T(2), length = size(N_iter, 1))
    eval_points = MyArray(range_re' .+ range_im * im)
end;
```

Differential equations

A dampened spring oscillator:

```
function harmonic_osc_eq(u, p, t)
    x, v = u[1], u[2]
    k, m, c = p[1], p[2], p[3]
    dx_dt = v
    dv_dt = - k/m * x - c/m * v
    du_dt = [dx_dt, dv_dt]
    return du_dt
end;
```

Spring constant k, mass m, friction coefficient c:

```
ho_pars = (k = 5.0, m = 1.0, c = 0.5);
```

```
x0, v0 = [1.0, 0.0];
```

```
t_span = (0.0, 15.0);
```

Solving the ODE

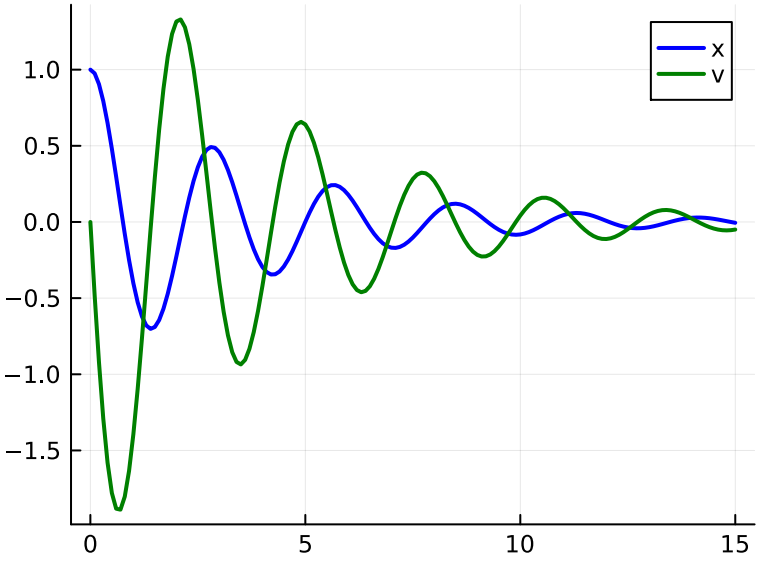
```
using OrdinaryDiffEq
```

sol =

	timestamp	value1	value2
1	0.0	1.0	0.0
2	0.1	0.975513	-0.483652
3	0.2	0.904843	-0.920226
4	0.3	0.793672	-1.29081
5	0.4	0.649377	-1.58064
6	0.5	0.48058	-1.77956
7	0.6	0.296675	-1.88235
8	0.7	0.107329	-1.8887
9	0.8	-0.0779958	-1.80302
10	0.9	-0.25049	-1.63394
	more		

```
sol = solve(
    ODEProblem(harmonic_osc_eq, [x0, v0], t_span, [ho_pars...]),
    saveat = 0:0.1:15
)
```

Plotting the ODE solution




```
let t = sol.t, x = sol[1, :], v = sol[2, :]
    plot(t, x, label="x", linecolor = :blue, linewidth=2)
    plot!(t, v, label="v", linecolor = :green, linewidth=2)
end
```

Fitting measured data

A statistical forward model:

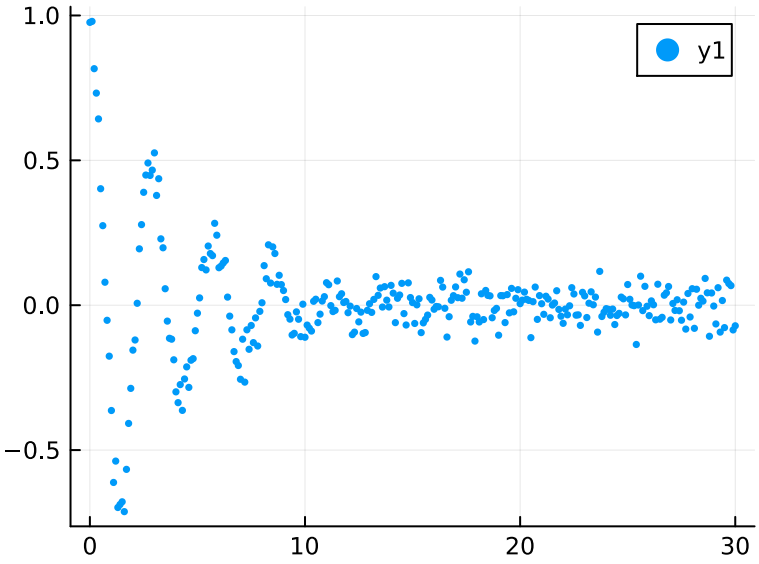
```
function fwd_model(t_obs::AbstractVector{<:Real}, pars::NamedTuple)
    m = 1.0
    (;x0, v0, k, c) = pars
    sol = solve(
        ODEProblem(
            harmonic_osc_eq, [x0, v0],
            (first(t_obs), last(t_obs)), [k, m, c]
        ),
        # sensealg = SciMLSensitivity.EnzymeVJP(),
        saveat = t_obs
    )
    x_expected = sol[1, :]
    σ_noise = 0.05
    return MvNormal(x_expected, σ_noise)
end;
```

Toy data generation

```
t_obs = 0:0.1:30;
```

```
p_truth = (x0 = 1.0, v0 = 0.0, k = 5.0, c = 0.5);
```

```
x_obs = rand(fwd_model(t_obs, p_truth));
```




```
scatter(t_obs, x_obs, ms = 2, msw = 0)
```

Likelihood definition

```
using DensityInterface, MeasureBase
```

```
ℒ = Likelihood(p -> fwd_model(t_obs, p), x_obs);
```

```
p_init = (x0 = 1.2, v0 = 0.1, k = 4.7, c = 1.0);
```

```
9.908821640448423
```

```
logdensityof(ℒ, p_init)
```

Maximum Likelihood fit

```
p_ctor = NamedTuple{(:x0, :v0, :k, :c)}
```

```
p_ctor = NamedTuple{propertynames(p_init)}
```

```
f_opt = logdensityof(ℒ) ∘ p_ctor;
```

```
using Optim
```

```
opt_result = Optim.maximize(log ∘ ℒ ∘ p_ctor, collect(p_init));
```

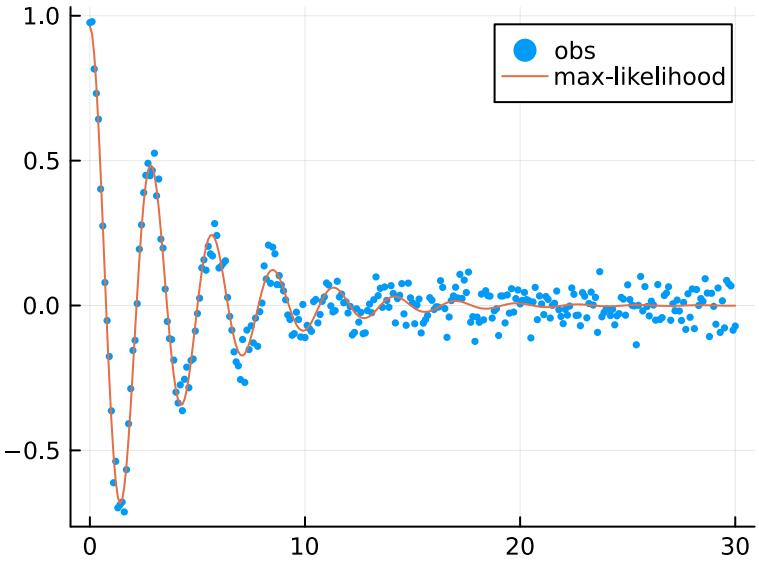
```
* Status: success

* Candidate solution
   Final objective value:     -4.761590e+02

* Found with
   Algorithm:     Nelder-Mead

* Convergence measures
   √(Σ(yᵢ-ȳ)²)/n ≤ 1.0e-08

* Work counters
   Seconds run:   0  (vs limit Inf)
   Iterations:    177
   f(x) calls:    317
```

```
1 opt_result.res
```

Maximum-Likelihood fit result

```
p_max_likelihood =   (x0 = 0.963757, v0 = -0.0231033, k = 4.97661, c = 0.484537)
```

```
1 p_max_likelihood = p_ctor(Optim.maximizer(opt_result))
```



Parameter uncertainly and correlation estimate

```
1 using ForwardDiff
```

```
p_Σ = 4×4 Matrix{Float64}:
       -0.000446878   0.000452095   0.000124725  -0.000216638
        0.000452095  -0.00297966   -0.00138582    0.000234806
        0.000124725  -0.00138582   -0.00126696    1.72752e-5
       -0.000216638   0.000234806   1.72752e-5   -0.000226908
```

```
1 p_Σ = inv(ForwardDiff.hessian(f_opt, Optim.maximizer(opt_result)))
```

Can also use reverse-mode AD:

```
import SciMLSensitivity
inv(Zygote.hessian(f_opt, Optim.maximizer(opt_result)))
```

Bayesian maxmium posterior (MAP)

```
using BAT
```

```
prior = BAT.distprod((
    x0 = Normal(0, 1), v0 = Normal(0, 1), k = Exponential(3.0), c = Exponential(0.5)
));
```

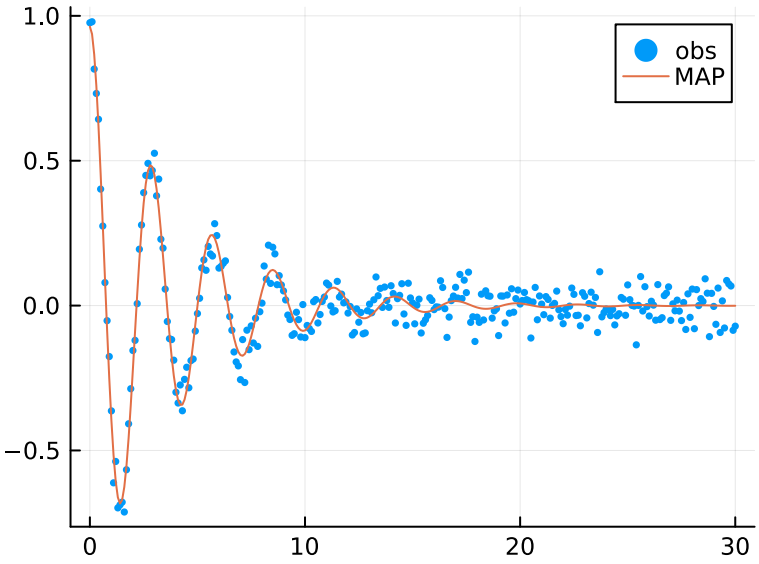
```
posterior = lbqintegral(ℒ, prior);
```

p_max_postior = (x0 = 0.962928, v0 = -0.0226242, k = 4.97635, c = 0.483874)

```
p_max_postior = bat_findmode(posterior, BATContext()).result
```

```
f_tol is deprecated. Use f_abstol or f_reltol instead. The provided value (0.0) wi
ll be used as f_reltol.
```

MAP fit result




```
begin
scatter(t_obs, x_obs, ms = 2, msw = 0, label = "obs")
plot!(t_obs, mean(fwd_model(t_obs, p_max_postior)), label = "MAP")
end
```

An incomplete tour of the Julia package ecosystem

Math

- ApproxFun.jl: Powerful function approximations
- FFTW.jl: Fast fourier transforms via FFTW
- DifferentialEquations.jl: A suite for numerically solving differential equations
- ... many many many more ...

Optimization

- JuMP.jl: Modeling language for Mathematical Optimization
- NLopt.jl: Optimization via NLopt
- Optim: Julia native nonlinear optimization

TypedTables and DataFrames

- Tables.jl: Abstract API for tabular data
- DataFrames.jl: Python/R-like dataframes
- TypedTables.jl: Type-stable tables
- Query.jl LINQ-inspired data query and transformation

Plotting and Visualization

- IJulia.jl: Julia Jupyter kernel
- Images.jl: Image processing
- PyPlot.jl: Use matplotlib/PyPlot from Julia
- Makie.jl: Hardware-accelerated plotting
- Plots.jl: Plotting with generic recipes and multiple backends

Statistics

- Distributions.jl: Probability distributions and associated functions
- StatsBase.jl: Statistics, histograms, etc.
- Turing.jl: Probabilistic model inference
- BAT.jl: Bayesian analysis toolkit
- Many, many specialized packages

Automatic Differentiation

- Meta-packages DifferentiationInterface.jl and AutoDiffOperators.jl
- ForwardDiff.jl: Forward-mode automatic differentiation
- Zyote.jl: Source-level reverse-mode automatic differentiation
- Enzyme.jl: LLVM-level reverse-mode automatic differentiation
- Mooncake.jl (WIP): Source-level reverse AD
- Several other packages available (ReverseDiff.jl, Nabla.jl, Yota.jl, ...)
- Exciting developements to come with new Julia Compiler features

Machine learning

- Lux.jl
- SimpleChains.jl
- Flux.jl: Julia native deep learning library
- ...

Orthogonal to GPU-Support and automatic differentiation (unlike Python ML)

Calling code in other languages

- Can natively call plain-C and Fortran code without overhead
- CxxWrap.jl: Wrap C++ packages for Julia (used by ROOT.jl and Geant4.jl)
- PyCall.jl and PythonCall.jl: Call Python from Julia
- RCall.jl: Call R from Julia
- MathLink.jl: Mathematica/Wolfram Engine integration
- ...

Efficient memory layout

- ArraysOfArrays.jl: Duality of flat and nested arrays
- StructArrays.jl, TypedTables.jl: AoS and SoA duality
- ValueShapes.jl: Duality of flat and nested structures
- ...

GPU Programming

- AMDGPU.jl: Julia on AMD GPUs (WIP)
- CUDA.jl: Julia on NVIDIA GPUs
- Metal.jl: Julia on Apple M-series GPUs

- oneAPI.jl: Julia on Intel oneAPI
- Experimental work on other accelerator platforms (e.g. Graphcore IPUs)

IDEs

- julia-vscode: Excellent Julia support in Visual Studio Code
- Plugins for many other code editors

Learning Julia

- Official "Getting started with Julia" site
- Julia language documentation
- The Fast Track to Julia (cheat sheet)
- Intro to Julia, by Jane Herriman (video)
- Modern Julia Workflows (online book)
- Think Julia, by Ben Lauwens (online book)

Summary

- Julia is productive, fast and fun - give it a chance!
- Multiple dispatch opens up powerful ways of combining code
- Try it out!