

A Journey Through



A DYNAMIC AND FAST LANGUAGE

THIBAUT CUVELIER

17 NOVEMBER, 2016

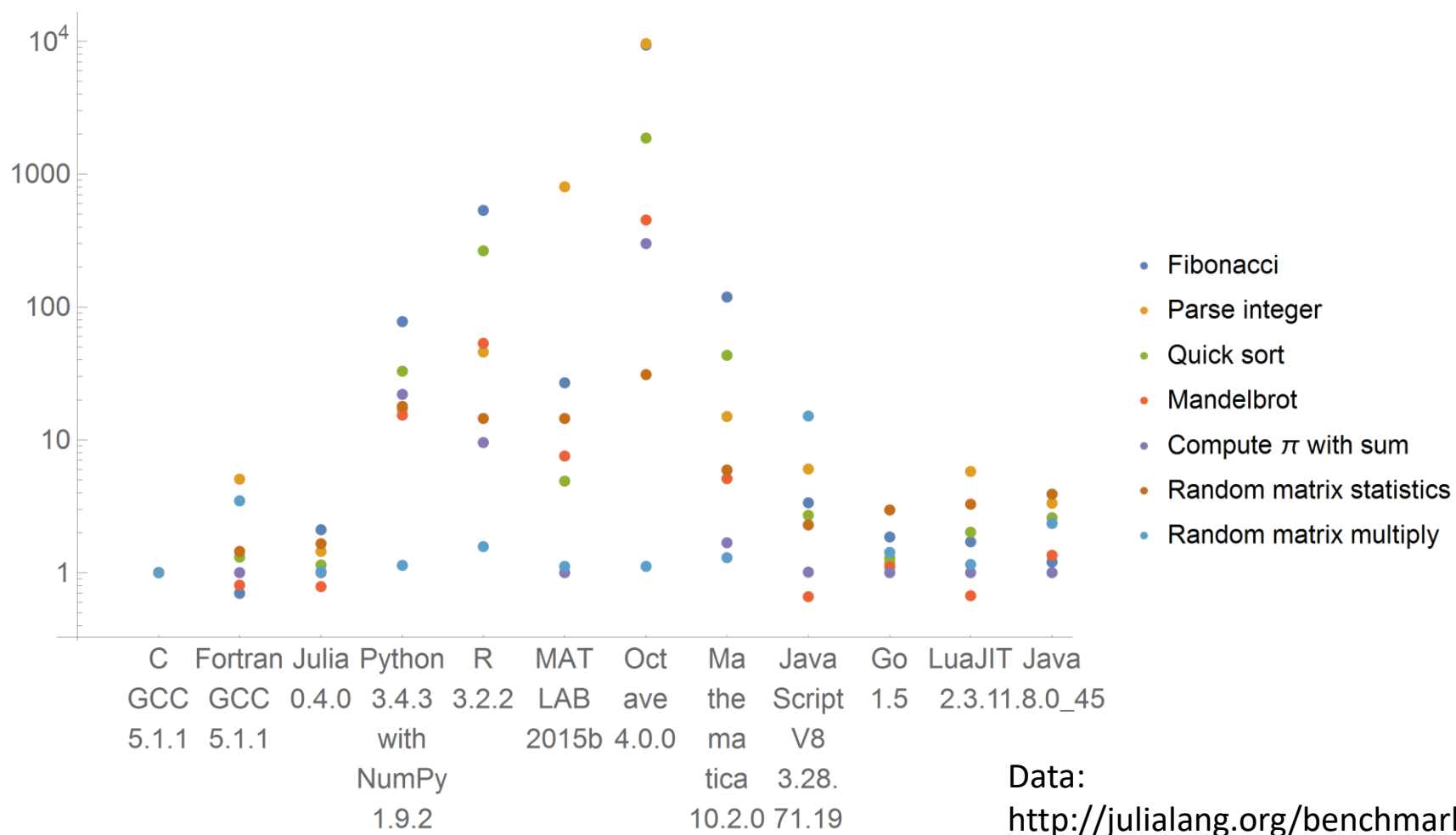
What is ?

- A programming language
 - For scientific computing first: running times are important!
 - But still dynamic, “modern”... and extensible!
- Often compared to MATLAB, with a similar syntax...
 - ... but much faster!
 - ... without the need for compilation!
 - ... with a large community!
 - ... and free (MIT-licensed)!

How fast is julia?

Comparison of run time between several languages and C

Times slower than C



How to install julia?

- Website: <http://julialang.org/>
- IDEs?
 - [Juno: Atom with Julia extensions](#)
 - Install Atom: <https://atom.io/>
 - Install Juno: in Atom, **File > Settings > Install**, search for **uber-juno**
 - [JuliaDT: Eclipse with Julia extensions](#)
- Notebook environment?
 - IJulia (think IPython)

Notebook environment

- The default console is not the sexiest interface
 - The community provides better ones!
- **Purely online, free: JuliaBox**
 - <https://juliabox.com/>
- Offline, based on Jupyter (still in the browser): IJulia
 - Install with:

```
julia> Pkg.add("IJulia")
```
 - Run with:

```
julia> using IJulia; notebook()
```

Contents of this presentation

- Core concepts
- Julia community
- Plotting
- Mathematical optimisation
- Data science
- Parallel computing
 - Message passing (MPI-like)
 - Multithreading (OpenMP-like)
 - GPUs
- Concluding words

Core concepts

What makes Julia dynamic?

- Dynamic type system with type inference
 - Multiple dispatch (see later)
 - But static typing is preferable for performance
- Macros to generate code on the fly
 - See later
- Garbage collection
 - Automatic memory management
 - No destructors, memory freeing
- Shell (REPL)

Function overloading

- A function may have multiple implementations, depending on its arguments
 - One version specialised for integers
 - One version specialised for floats
 - Etc.

- In Julia parlance:
 - A **function** is just a name (for example, +)
 - A **method** is a “behaviour” for the function that may depend on the types of its arguments
 - `+(::Int, ::Int)`
 - `+(::Float32, ::Float64)`
 - `+(::Number, ::Number)`
 - `+(x, y)`

Function overloading: multiple dispatch

- All parameters are used to determine the method to call
 - C++'s virtual methods, Java methods, etc.: **dynamic** dispatch on the first argument, **static** for the others
 - Julia: **dynamic** dispatch on **all** arguments
- Example:
 - Class Matrix, specialisation Diagonal, with a function `add()`
 - `m.add(m2)`: standard implementation
 - `m.add(d)`: only modify the diagonal of `m`
 - What if the type of the argument is dynamic? Which method is called?

Function overloading: multiple dispatch

- What does Julia do?
- The user defines methods:
 - `add(::Matrix, ::Matrix)`
 - `add(::Matrix, ::Diagonal)`
 - `add(::Diagonal, ::Matrix)`
- When the function is called:
 - All types are **dynamically** used to choose the right method
 - Even if the type of the matrix is not known at compile time

Fast Julia code?

- First: Julia compiles the code before running it (JIT)
- To fully exploit multiple dispatch, write **type-stable** code
 - Multiple dispatch is slow when performed at run time
 - A variable should keep its type throughout a function
- If the type of a variable is 100% known, then the method to call is too
 - All code goes through JIT before execution

Object-oriented code?

- Usual syntax makes little sense for mathematical operations
 - `+(::Int, ::Float64)`: belongs to `Int` or `Float64`?
- Hence: syntax very similar to that of C
 - `f(o, args)` instead of `o.f(args)`
- However, Julia has:
 - A type hierarchy, including **abstract** types
 - Constructors

Community and packages

A vibrant community

- Julia has a large community with many extension packages available:
 - For plotting: Plots.jl, Gadfly, Winston, etc.
 - For graphs: Graphs.jl, LightGraph.jl, Graft.jl, etc.
 - For statistics: DataFrames.jl, Distributions.jl, TimeSeries.jl, etc.
 - For machine learning: JuliaML, ScikitLearn.jl, etc.
 - For Web development: Mux.jl, Escher.jl, WebSockets.jl, etc.
 - For mathematical optimisation: JuMP.jl, Convex.jl, Optim.jl, etc.
- A list of all **registered** packages: <http://pkg.julialang.org/>

Package manager

- How to install a package?

```
julia> Pkg.add("PackageName")
```

- No .jl in the name!

- Import a package (from within the shell or a script):

```
julia> import PackageName
```

- How to remove a package?

```
julia> Pkg.rm("PackageName")
```

- All packages are hosted on GitHub

- Usually grouped by interest: JuliaStats, JuliaML, JuliaWeb, JuliaOpt, JuliaPlots, JuliaQuant, JuliaParallel, JuliaMaths...
- See a list at <http://julialang.org/community/>

Plots

Creating plots: Plots.jl

- Plots.jl: an interface to multiple plotting engines (e.g. GR or matplotlib)
- Install the interface and one plotting engine (GR is fast):

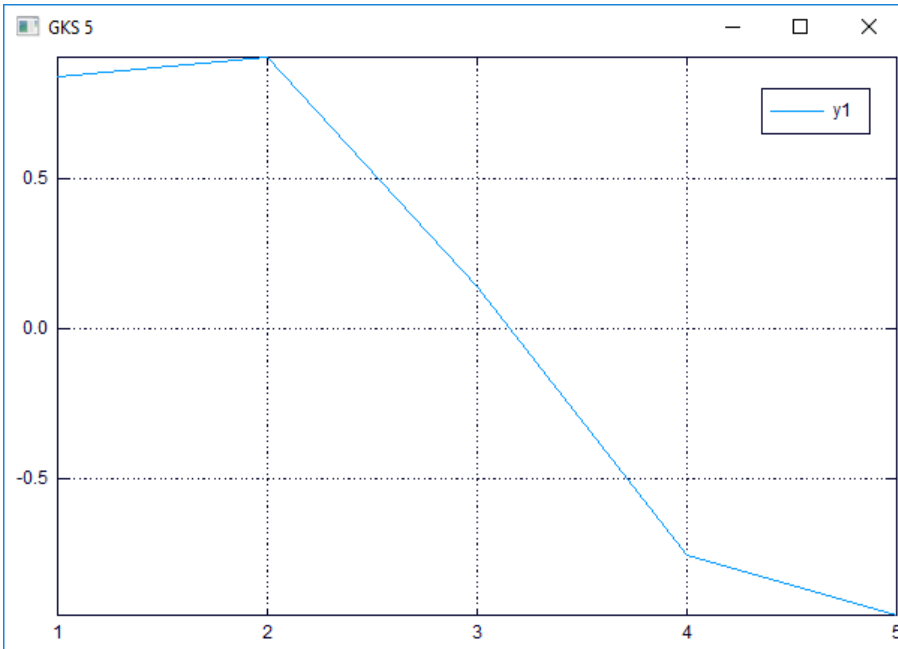
```
julia> Pkg.add("Plots")  
julia> Pkg.add("GR")  
julia> using Plots
```

- Documentation: <https://juliaplots.github.io/>

Basic plots

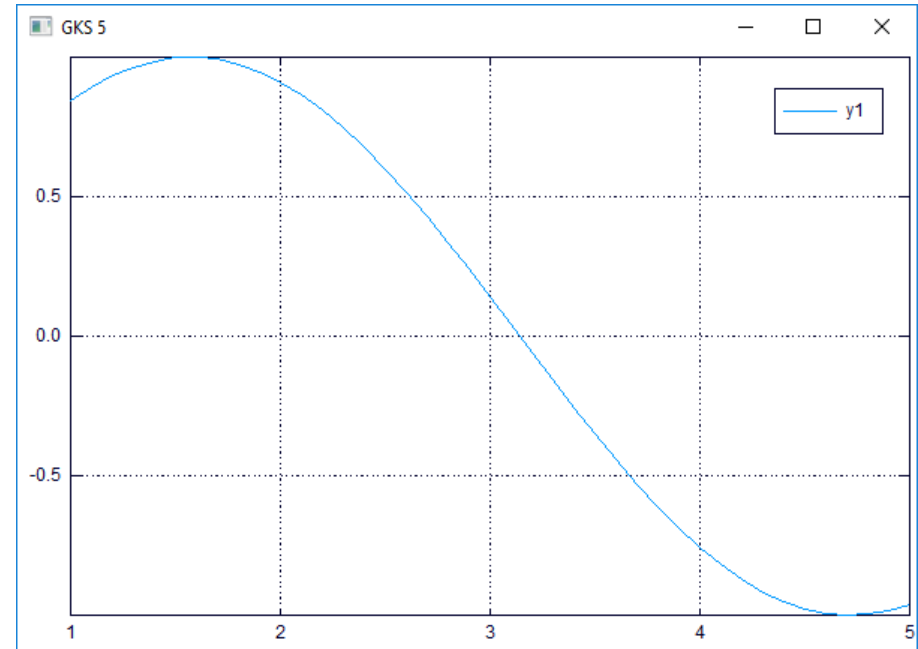
- Basic plot:

```
julia> plot(1:5, sin(1:5))
```



- Plotting a mathematical function:

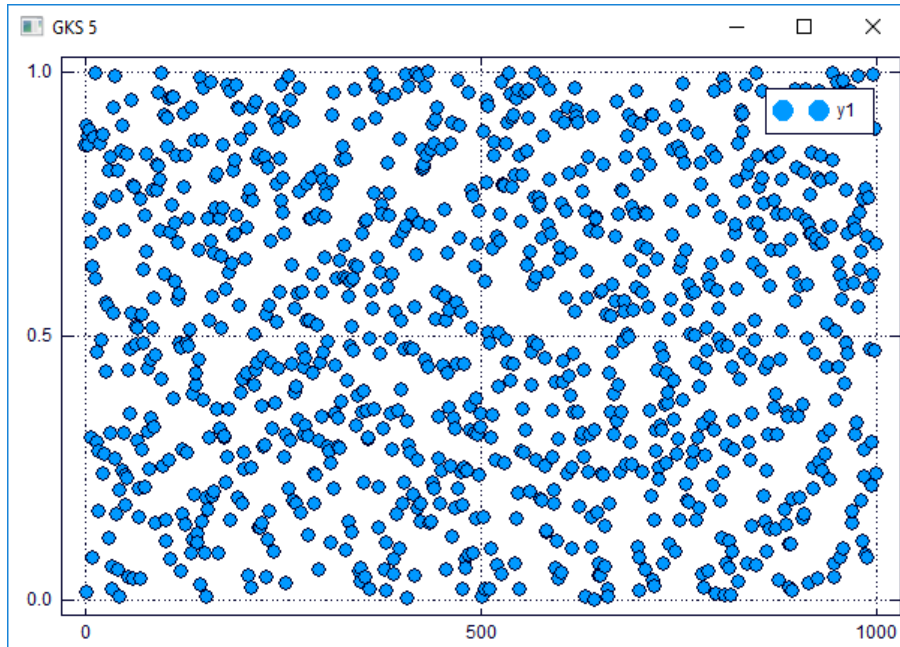
```
julia> plot(sin, 1:.1:5)
```



More plots

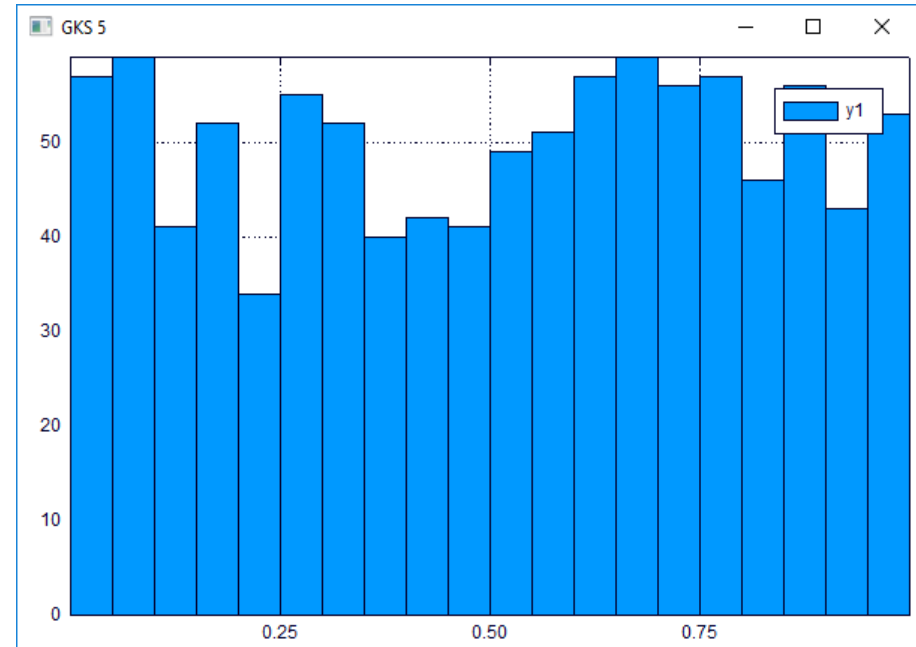
- Scatter plot:

```
julia> scatter(rand(1000))
```



- Histogram:

```
julia> histogram(rand(1000),  
nbins=20)
```



Mathematical optimisation

AND MACROS!

Mathematical optimisation: JuMP

- JuMP provides an easy way to translate optimisation programs into code
- First: install it along with a solver

```
julia> Pkg.add("JuMP")  
julia> Pkg.add("Cbc")  
julia> using JuMP
```

$$\begin{aligned} &\max x + y \\ &\text{s.t. } 2x + y \leq 8 \\ &\quad 0 \leq x \leq +\infty \\ &\quad 1 \leq y \leq 20 \end{aligned}$$

```
m = Model()  
@variable(m, x >= 0)  
@variable(m, 1 <= y <= 20)  
@objective(m, Max, x + y)  
@constraint(m, 2 * x + y <= 8)  
solve(m)
```

Behind the nice syntax: macros

- Macros are a very powerful mechanism
 - Much more powerful than in C or C++!
- Macros are function
 - Argument: Julia code
 - Return: Julia code
- They are the main mechanism behind JuMP's syntax
 - Easy to define DSLs in Julia!
 - Example:
<https://github.com/JuliaOpt/JuMP.jl/blob/master/src/macros.jl#L743>
- How about speed?
 - JuMP is as fast as a dedicated compiler (like AMPL)
 - JuMP is much faster than Pyomo (similar syntax, but no macros)

Data science

Data frames: DataFrames.jl

- R has the data frame type: an array with named columns

```
df = DataFrame(N=1:3, colour=["b", "w", "b"])
```

- Easy to retrieve information in each dimension:

```
df[:colour]  
df[1, :]
```

- The package has good support in the ecosystem
 - Easy plot with Plots.jl: just install StatPlots.jl, it just works
 - Understood by machine learning packages, etc.

Data selection: Query.jl

- SQL is a nice language to query information from a data base: select, filter, join, etc.
- C# has a similar tool integrated into the language (LINQ)
- Julia too, with a syntax inspired by LINQ: Query.jl
- On data frames:

```
@from i in df begin
    @where i.N >= 2
    @select {i.colour}
    @collect DataFrame
end
```

Machine learning

- Many tools to perform machine learning
- A few to cite:
 - [JuliaML](#): generic machine learning project, highly configurable
 - [GLM](#): generalised linear models
 - [Mocha](#): deep learning (similar to Caffe in C++)
 - [ScikitLearn](#): uniform interface for machine learning

Parallel programming

MULTITHREADING

MESSAGE PASSING

ACCELERATORS

Message passing

- Multiple machines (or processes) communicate over the network
 - For scientific computing: like MPI
 - For big data: like Hadoop (close to message passing)
- The Julia way?
 - Similar to MPI... but useable
 - Only one side manages the communication

Message passing

- Two primitives:
 - `r = @spawn`: start to compute something
 - `fetch(r)`: retrieve the results of the computation
 - Start Julia with `julia -p 2` for two processes on the current machine
- Example: generate a random matrix on another machine (#2), retrieve it on the main node

```
r = @spawn 2 rand(2, 2)
fetch(r)
```

Message passing: reductions

- Hadoop uses the map-reduce paradigm
- Julia has it too!
- Example: flip a coin multiple times and count heads

```
nheads = @parallel (+) for i in 1:500
    Int(rand(Bool))
end
```

Multithreading

- New (and experimental) with Julia 0.5: multithreading
- Current API (not set in stone):
 - `@Threads.threads` before a loop
 - As simple as MATLAB's `parfor` or OpenMP!
- Add the environment variable `JULIA_NUM_THREADS` before starting Julia

Multithreading

```
array = zeros(20)
@Threads.threads for i in 1:20
    array[i] = Threads.threadid()
end
```

GPU computing: ArrayFire.jl

- GPGPU is a hot topic currently, especially for deep learning
 - Use GPUs to perform computations
 - Many cores available (1,000s for high-end ones)
 - Very different architecture
- ArrayFire provides an interface for GPUs and other accelerators:
 - Easy way to move data
 - Premade kernels for common operations
 - Intelligent JIT rewrites operations to use as few kernels as possible
 - For example, linear algebra: $\mathbf{A} \mathbf{b} + \mathbf{c}$ in one kernel
- Note: CUDA offloading will probably be included in Julia
<https://github.com/JuliaLang/julia/issues/19302>
Similar to OpenMP offloading

GPU computing

- Installation:
 - First install the ArrayFire library:
<http://arrayfire.com/download/>
 - Then install the Julia wrapper:

```
Pkg.add("ArrayFire")
```
 - Load it:

```
using ArrayFire
```

GPU computing

- Ensure the OpenCL backend is used (or CUDA, or CPU):

```
setBackend(AF_BACKEND_OPENCL)
```

- Send an array on the GPU:

```
a_cpu = rand(Float32, 10, 10);  
a_gpu = AFArray(a_cpu);  
b_gpu = AFArray(rand(Float32, 10, 10));
```

- Then work on it as any Julia array:

```
c_gpu = a_gpu + b_gpu;
```

- Finally, retrieve the results:

```
c_cpu = Array(c_gpu);
```

Concluding remarks

And so... shall I use Julia?

- First drawback of Julia: no completely stable version yet
 - Syntax can still change (but not a lot)
 - Also for packages: nothing is really 100% stable

- Quite young: appeared in 2012
 - 0.5 in September 2016 (original plans: June 2016)
 - 0.6 in January 2017 (original plans: September 2016), 1.0 just after

- ... but likely to survive!
 - Enterprise backing the project: JuliaComputing
 - 7 books about Julia (5 in 2016)

- Not ready for production... yet