# A Journey Through

**julia**

A DYNAMIC **AND** FAST LANGUAGE

THIBAUT CUVELIER
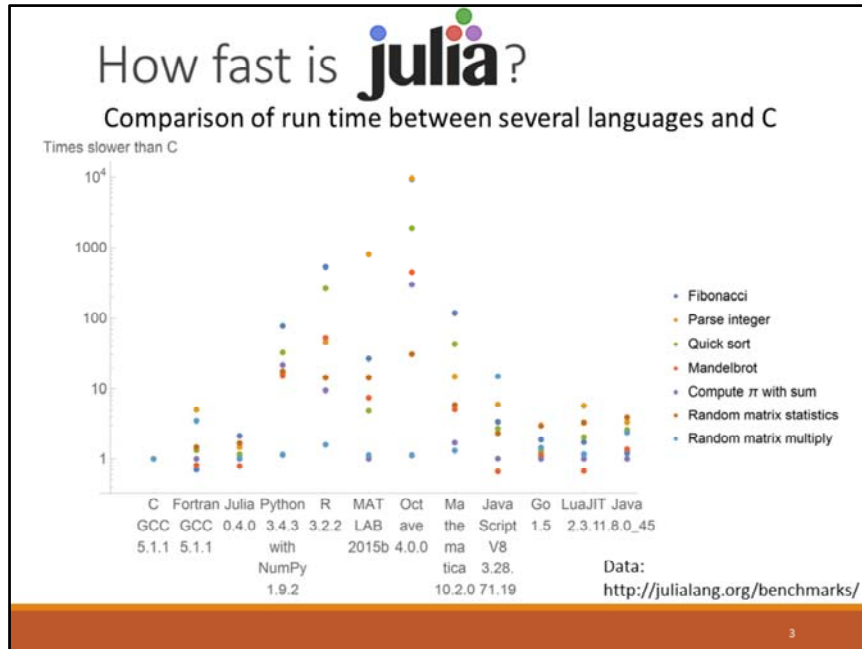
17 MAY, 2017

◆IEEE

1

Actually, Julia solves the two-language problem: no need for one nice language (such as Python or R) and one fast language (like C or Fortran). The whole code can be written in the same language.

Compilation: unlike C or C++ or Fortran… or MATLAB's MEX.

Can reach performance of C or Fortran code!
Performance relative to C.

OK, that is becoming old (Julia 0.6 is being released).

# 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

  - Also: JuliaPro, a Julia distribution "with batteries included"
    - https://juliacomputing.com/products/juliapro.html

- Notebook environment?
  - IJulia (think IPython)

4

# 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

- Basic syntax

- Core concepts

- Julia community
  ◦ Plotting
  ◦ Image processing
  ◦ Mathematical optimisation
  ◦ Data science

- Parallel computing
  ◦ Message passing (MPI-like)
  ◦ Multithreading (OpenMP-like)
  ◦ GPUs

- Concluding words

6

Basic syntax

Julia's syntax is quite intuitive: very similar to that of MATLAB.
For those who come from other languages: mostly, blocks start with a keyword and end with end.

If you need a syntax help: http://cheatsheets.quantecon.org/

## A small taste of Julia

```julia
# Solves 0 = a x² + b x + c for x
function quad(a::Float64, b::Float64, c::Float64)
    discriminant = b^2 - 4a * c
    r1 = (- b + sqrt(discriminant)) / 2a
    r2 = (- b - sqrt(discriminant)) / 2a
    r1, r2
end
```

Particular parts of the syntax:
- Typing not necessary for variables or arguments, but still very useful!
- Can do products by a constant without operator: code more readable
- Automatically returns the last expression
- Can return more than one value for a function

The most important points when trying to better exploit the language.
How can Julia be fast?

# 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)

10

# 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)`

Integers, floating-point numbers, generic numbers (any kind of number), and… no type (if no other method applies).

# 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?

12

---

Virtual methods: dynamic dispatch is made on the object, not the other arguments (i.e. the type of `*this`).

Example: the standard implementation will be used, unfortunately (due to static dispatch).

# 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

Julia is as specific as possible: it uses the most precise method for the arguments at hand.
Here, for a diagonal matrix, no way to use the first method. But if a sparse matrix is defined and comes with no such + function, then Julia will have to rely on the first method.

## 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

If a variable can change type in a function, then Julia must use multiple dispatch at each operation.

JIT can choose the right method to call, instead of relying on multiple dispatch at run time. Hence: multiple dispatch performed at JIT time.

# 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

15

# Real macros

- C-like macros are limited
  - Only text replacement

- Julia macros: can rewrite code!
  - Function that takes code in argument and outputs code

- Basic example:
```
julia> @show sqrt(complex(-1.))
sqrt(complex(-1.0)) = 0.0 + 1.0im
0.0 + 1.0im
```

# Real macros

- Building block for DSLs, e.g. for differential equations:

```
g = @ode_def LorenzExample begin
    dx = σ*(y-x)
    dy = x*(ρ-z) - y
    dz = x*y - β*z
end σ=>10.0 ρ=>28.0 β=(8/3)
u0 = [1.0;0.0;0.0]
tspan = (0.0,1.0)
prob = ODEProblem(g,u0,tspan)
sol = solve(prob)
```

Details for the packages needed for this:
http://docs.juliadiffeq.org/stable/tutorials/ode_example.html#Defining-Systems-of-Equations-Using-ParameterizedFunctions.jl-1

Community and packages

Julia has a truckload of packages, and that is a great selling point for the language: its community.

## 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/

19

Package directory: list many existing packages (not all, often due to development in progress).
Directly used for the package manager.

## 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/

20

Other interest groups:
- Quantitative Economics: http://quant-econ.net/jl/getting_started.html
- JuliaGPU, BioJulia, JuliaAstro, etc.

Plots

For scientific applications, plots are a must-have to graphically represent data, an algorithm's behaviour, etc.

Julia has quite a few native rendering engines, but also packages that allow using existing plotting engines (such as Matplotlib or GR).

It also has one common interface for those many plotting engine, which is the topic now.
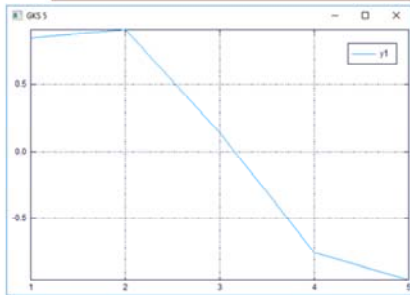
Quit PowerPoint, move on to Ijulia, show how the plots integrate into the interface.

# Basic plots

- Basic plot:

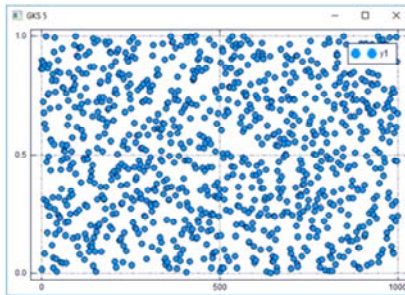  `julia> plot(1:5, sin(1:5))`

- Plotting a mathematical function:
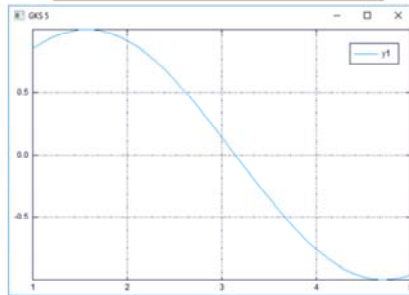
  `julia> plot(sin, 1:.1:5)`

Albeit made for scientific computations, Julia is also open to the Web.
Lightweight Web frameworks, Web servers… and frameworks to build Web UIs.

# Basic processing: Images.jl and family

- Part of the JuliaImages organisation

- MATLAB-like interface, with many features:
  - Filtering: ImageFiltering
  - Feature extraction: ImageFeatures
  - Not full yet

- A repository of test images: TestImages

- Standalone visualisation (outside IJulia): ImageView

26

# Show and process an image

```
using Images, TestImages, ImageView,
ImageFiltering

img = testimage("mandrill")

imshow(img)

imgg = imfilter(img, Kernel.gaussian(3))

imshow(imgg)
```
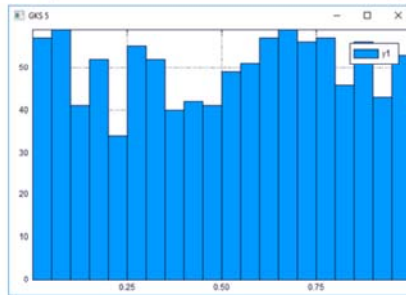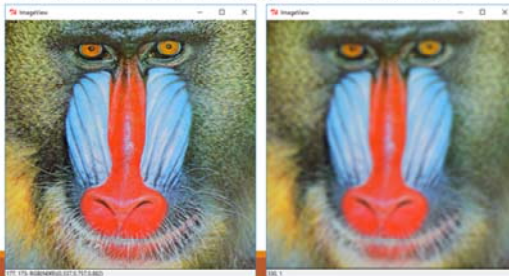
Albeit made for scientific computations, Julia is also open to the Web.
Lightweight Web frameworks, Web servers… and frameworks to build Web UIs.

# Web applications: Escher.jl

- Escher is a Web application framework
  - No need to use of HTML or anything: pure Julia
  - Based on the concept of **tiles**

- Predefined tiles:
  - Text (including Markdown and LaTeX)
  - Plots
  - Layouts (including tabs and pages)

- Main use case: provide a Web UI for a scientific application

- Similar to R's Shiny

- Documentation: http://escher-jl.org/

# Install and run Escher's server

- Escher comes with an integrated Web server
- Install Escher:
```
julia> Pkg.add("Escher")
```
- Run the Web server:
```
julia> using Escher
julia> include(Pkg.dir("Escher", "src", "cli", "serve.jl"))
julia> cd(Pkg.dir("Escher", "examples"))
julia> escher_serve()
```
  ◦ Here: started from within the examples
  ◦ Example: http://127.0.0.1:5555/user-guide

- Note: for Julia 0.5, you must check out the last version:
```
julia> Pkg.checkout("Escher")
```

30

Mathematical optimisation

AND MACROS!

Quite large community in optimisation. Multiple modelling layers, some optimisation solvers written in Julia (Optim.jl, Pajarito.jl), links to most existing solvers.

# Mathematical optimisation: JuMP

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

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

$$\max x + y$$
$$\text{s.t.}\, 2x + y \leq 8$$
$$0 \leq x \leq +\infty$$
$$1 \leq y \leq 20$$

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

Transition:

Quite specific topic, yes. But example of one selling point of Julia: macros. Look at the @ signs in the code!

Without these macros, the code makes little sense (x >= 0 when x has not yet been defined!?). However, with the macros, Julia does not directly evaluate the code: rather, JuMP rewrites it.

## 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)

33

Can a generic mechanism be close to a specific compiler? AMPL: dedicated programming language for optimisation, with specifically optimised compiler; it ought to be very fast.
Python has no macros, so Pyomo can only rely on function calls. And this is very slow.

Data science

Julia is also made for data treatment and machine learning, with packages inspired by R and Python.

# 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
```

An other example of DSL embedded within Julia, which makes queries really simple.

# 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

JuliaML: set of comprehensive packages containing many loss and penalty functions, data transformations, plotting, etc.
ScikitLearn: port of the Python library

For scientific computations… and big data (to name a few), parallel is needed: must solve very large problems, deal with enormous quantities of data.

Multiple paradigms so far: multithreading (cores of a machine), message passing (multiple machines), accelerators (GPUs). All three are currently supported within Julia.

# 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)
```

40

Note: can also be used with clusters, of course. Example of Slurm script:

```
#!/bin/bash
#SBATCH --job-name="juliaTest"
#SBATCH --output="juliaTest.%j.%N.out"
#SBATCH --partition=compute
#SBATCH --nodes=8
#SBATCH --export=ALL
#SBATCH --ntasks-per-node=24
#SBATCH -t 01:00:00
export SLURM_NODEFILE=`generate_pbs_nodefile`
./julia --machinefile $SLURM_NODEFILE test.jl
```

# 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

42

Should be finalised with Julia 1.0.

## Multithreading

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

Still experimental API, though.

## 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: **A b + c** in one kernel

- Note: CUDA offloading will probably be included in Julia
  https://github.com/JuliaLang/julia/issues/19302
  Similar to OpenMP offloading

44

High-end CPUs: up to 22 cores, with a price similar to a high-end GPU (for example, http://wccftech.com/intel-xeon-e5-2699a-v4-skylake-ep-2017-launch/). Architecture differences: GPU cores are grouped in streaming multiprocessors/compute units. All cores within this group perform the **same instruction** on **different data**.

Operations rewriting? A*x+b: not performed as c=A*b then d=c+b, but directly as A*b+c (using the corresponding kernel).

# 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

Here: mainly libraries around Julia (except for parallel programming). The goal is to keep the core language sleek, and to rely on packages for functionalities.

Ask the audience: What do **you** think of Julia?

## 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

48

Julia 0.6 feature freeze: end of 2016. Unlikely to be on time.