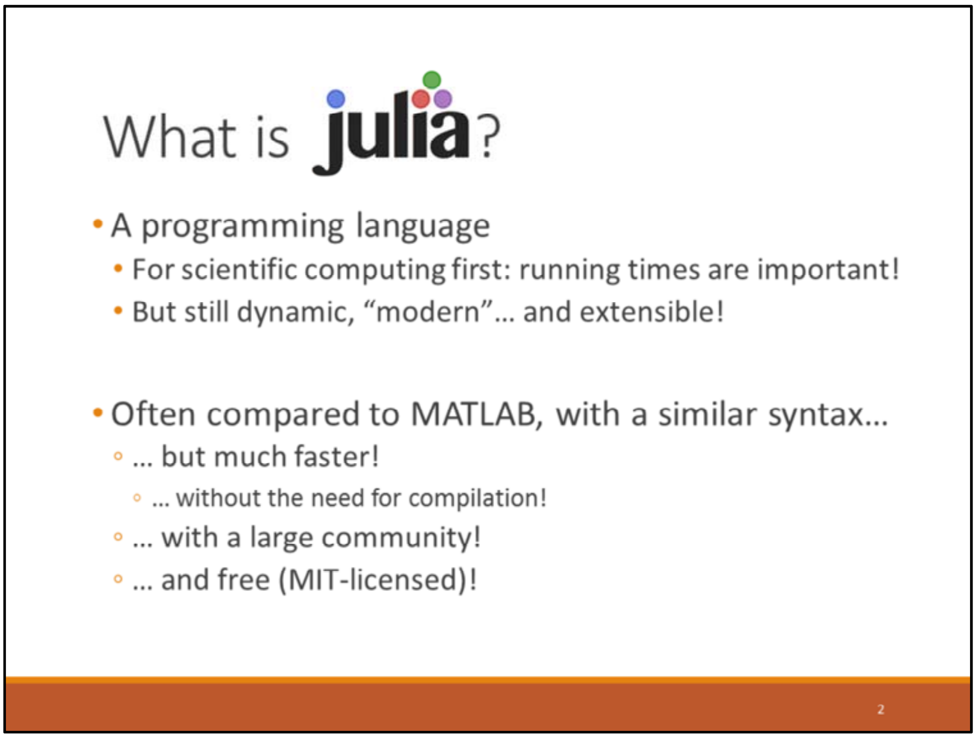
A Journey Through julia

A DYNAMIC **AND** FAST LANGUAGE

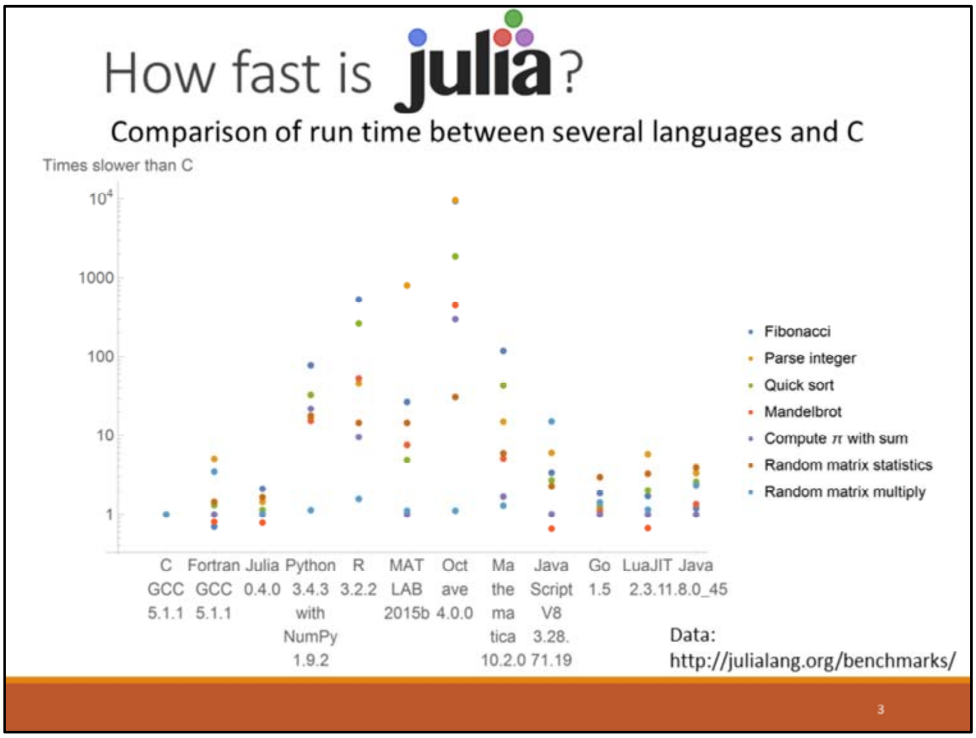
THIBAUT CUVELIER

17 NOVEMBER, 2016

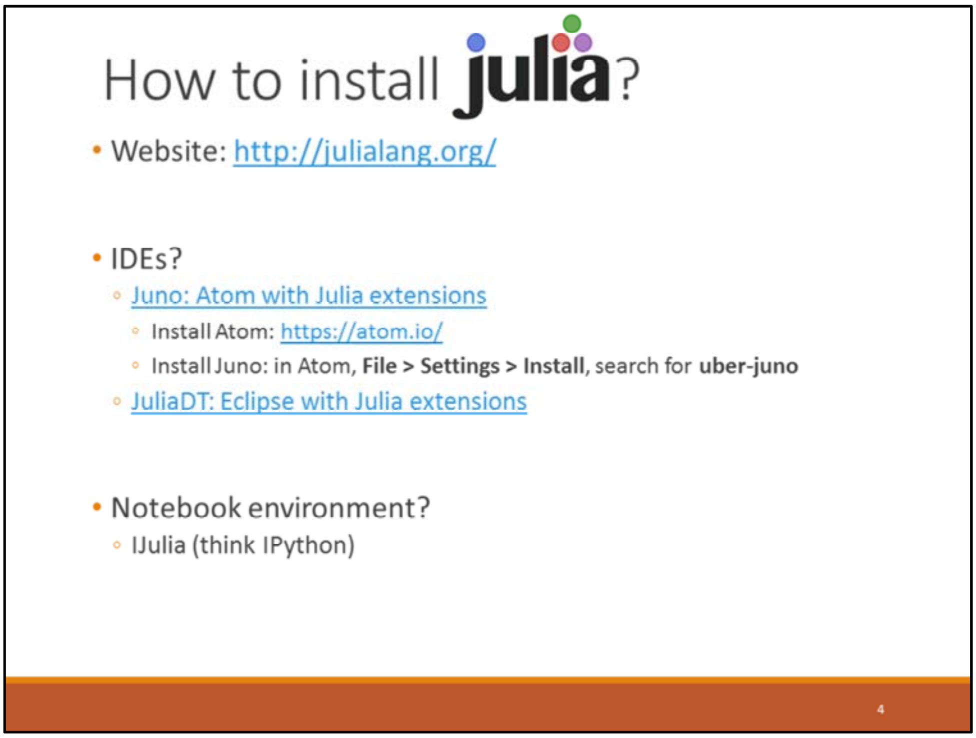


Actually, Julia solves the two-language problem: no need for one nice language (such as Python or R) and one fast language (like C or Fortran). The whole code can be written in the same language.

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



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


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)
4

Notebook environment

- The default console is not the sexiest interface
  - The community provides better ones!

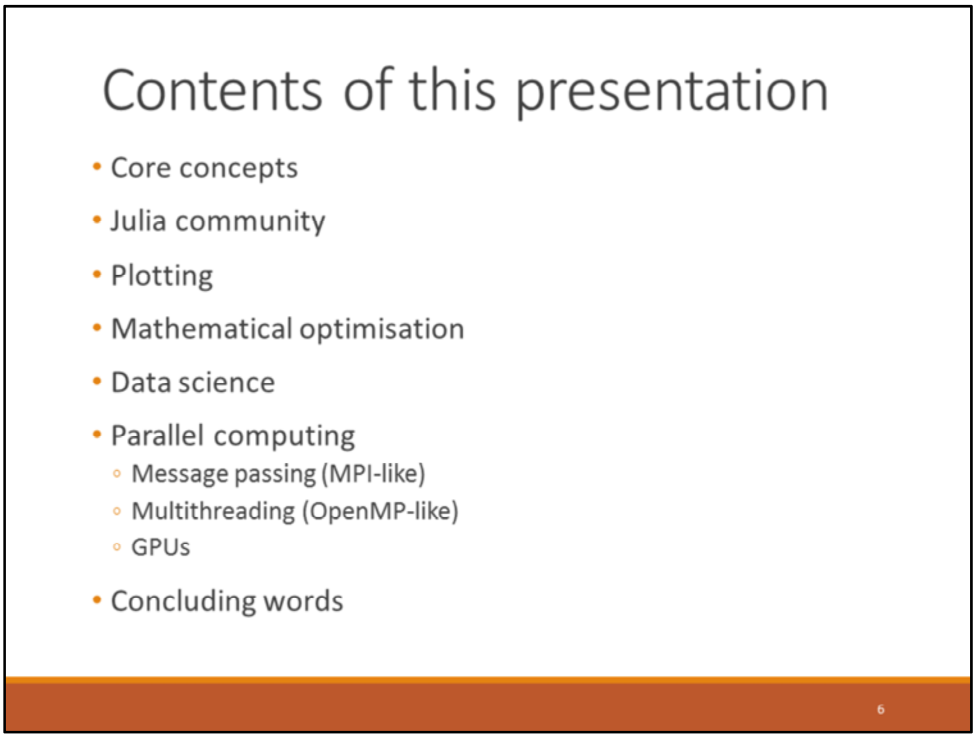
- **Purely online**, free: JuliaBox
  - https://juliabox.com/

- Offline, based on Jupyter (still in the browser): IJulia
  - Install with:

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

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

5



No syntax in this presentation: quite easy to get the basics (like many languages, except no curly braces: only keyword-end blocks).

Core concepts

Not really syntax, but the most important points when trying to better exploit the language.
How can Julia be fast?

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

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)`

9

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?

10

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).

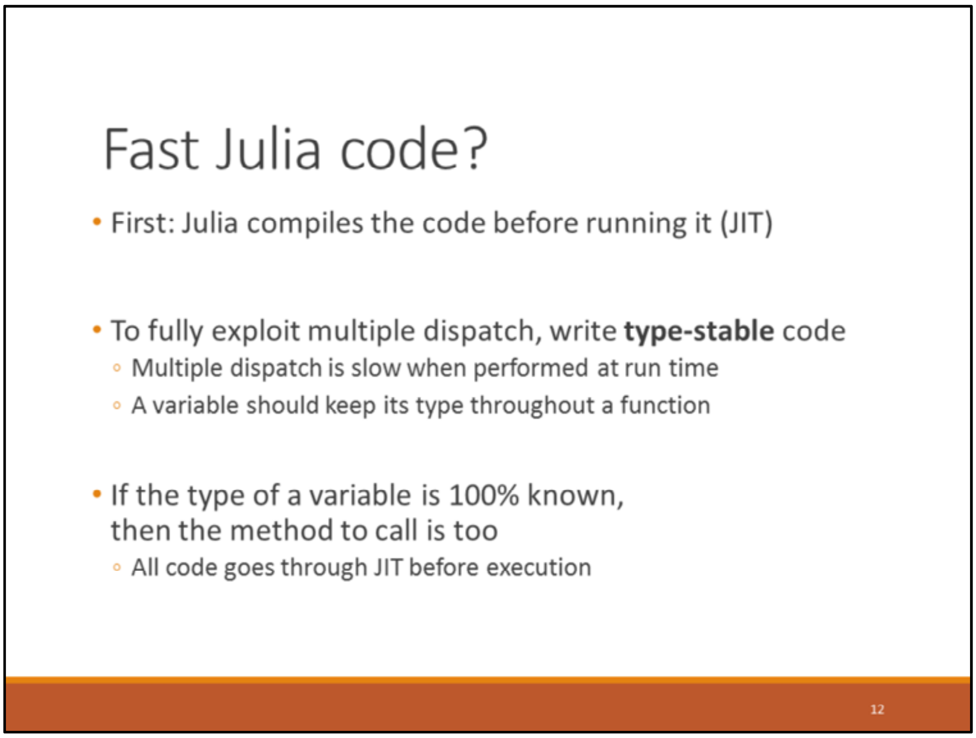
Function overloading: multiple dispatch

- 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

11

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.

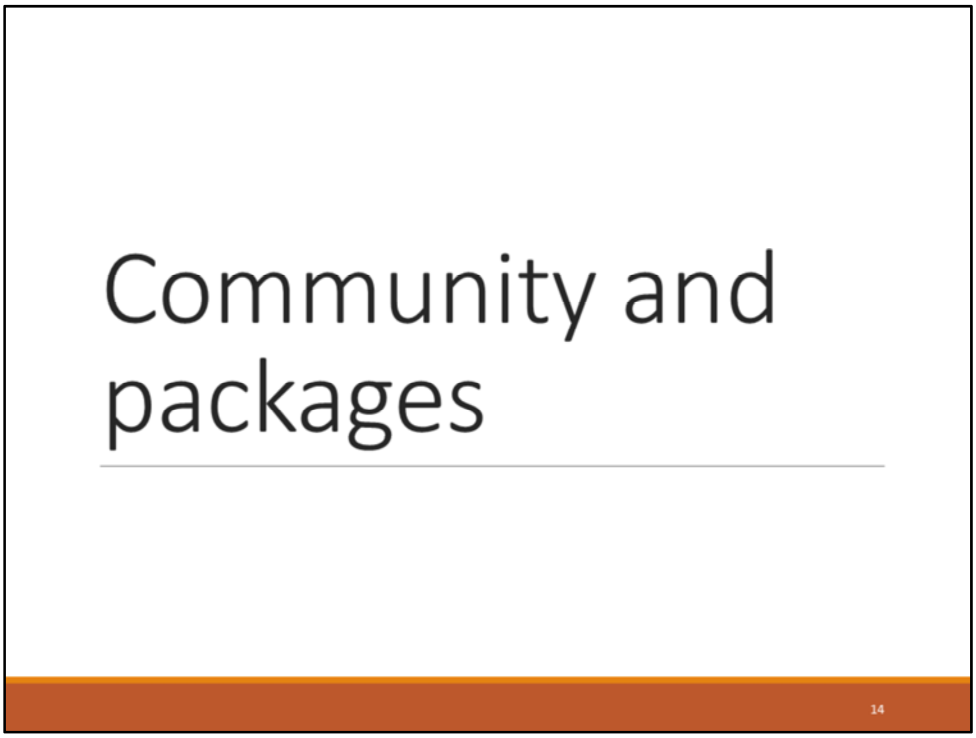


If a variable can change type in a function, then Julia must use multiple dispatch at each operation.

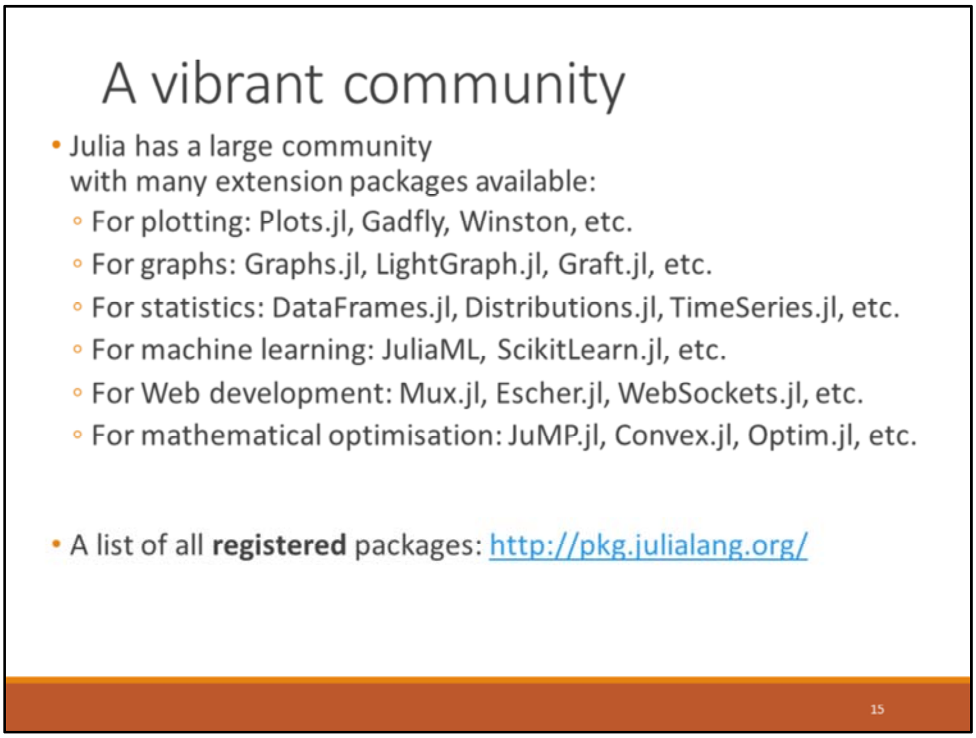
JIT can choose the right method to call, instead of relying on multiple dispatch at run time. Hence: multiple dispatch performed at JIT time.

Object-oriented code?

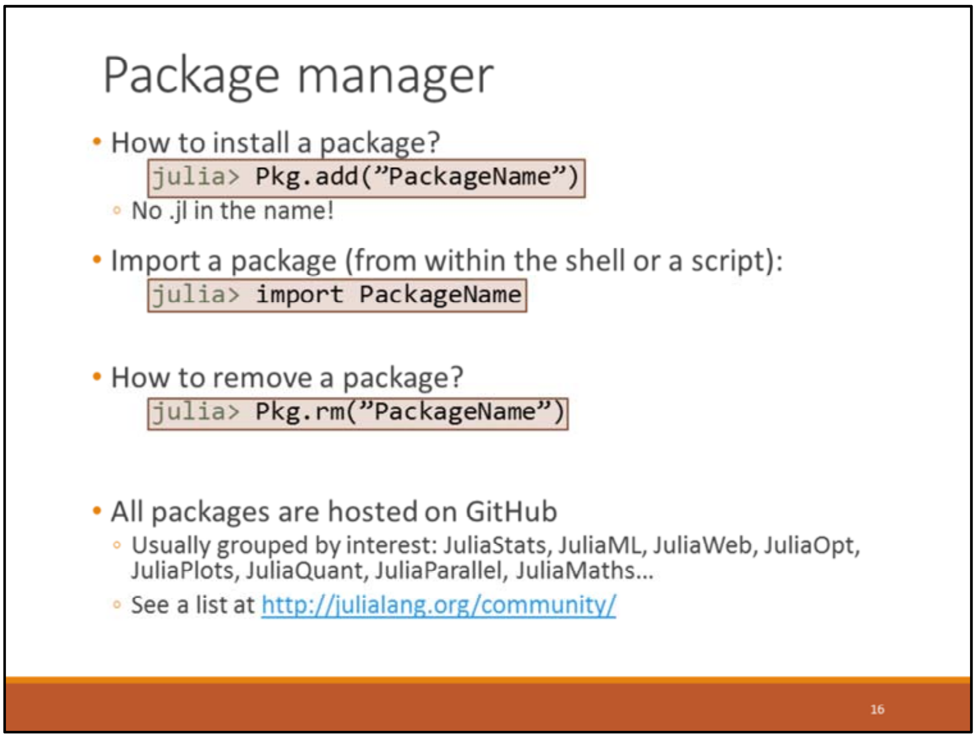
- Usual syntax makes little sense for mathematical operations
  - `+(::Int, ::Float64)`: belongs to Int or Float64?
- Hence: syntax very similar to that of C
  - `f(o, args)` instead of `o.f(args)`
- However, Julia has:
  - A type hierarchy, including **abstract** types
  - Constructors



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

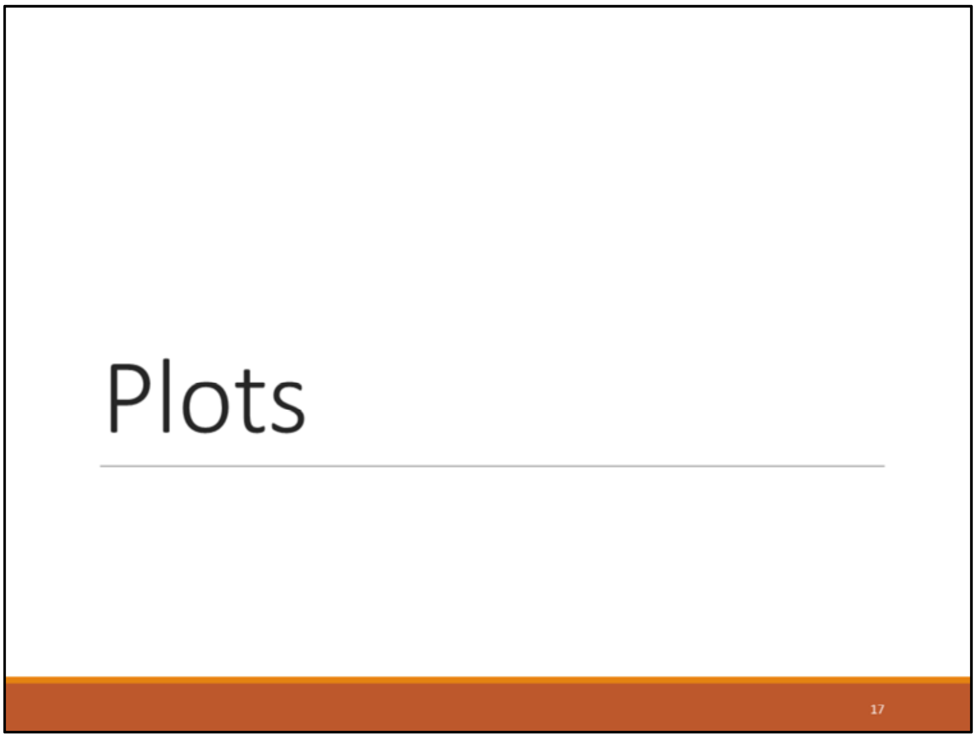


Package directory: list many existing packages (not all, often due to development in progress).
Directly used for the package manager.



Other interest groups:

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



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.

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/

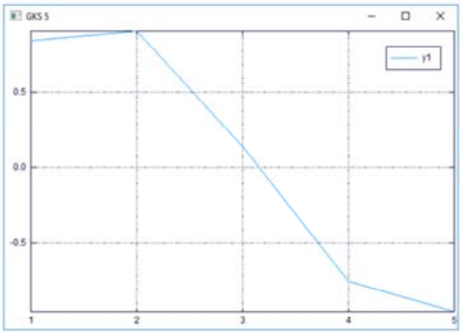
18

Quit PowerPoint, move on to Ijulia, show how the plots integrate into the interface.

Basic plots

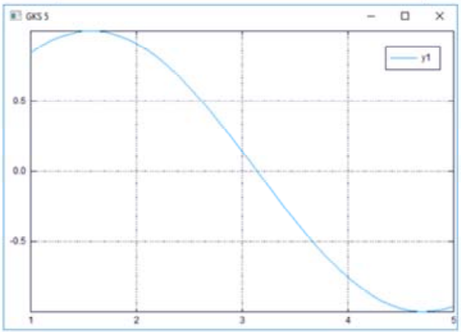
- Basic plot:

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



- Plotting a mathematical function:

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



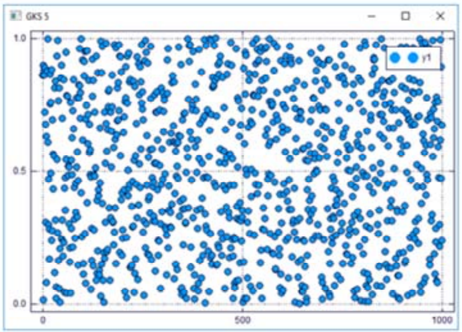
More plots

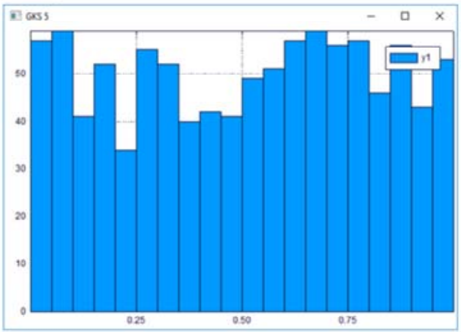
- Scatter plot:

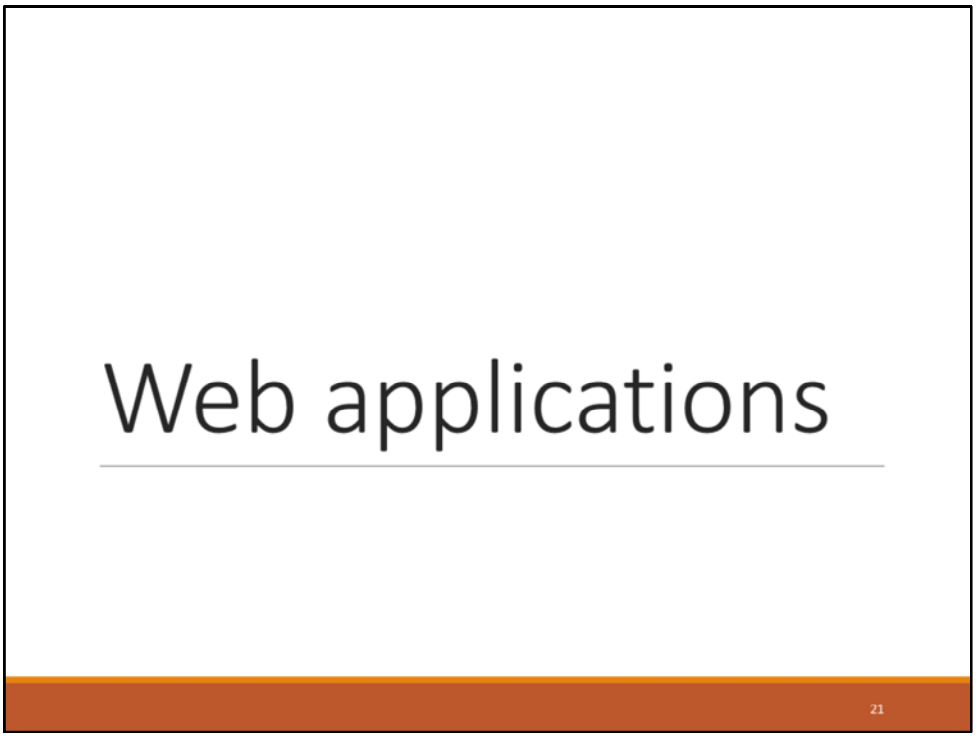
```
julia> scatter(rand(1000))
```

- Histogram:

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







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”)
```

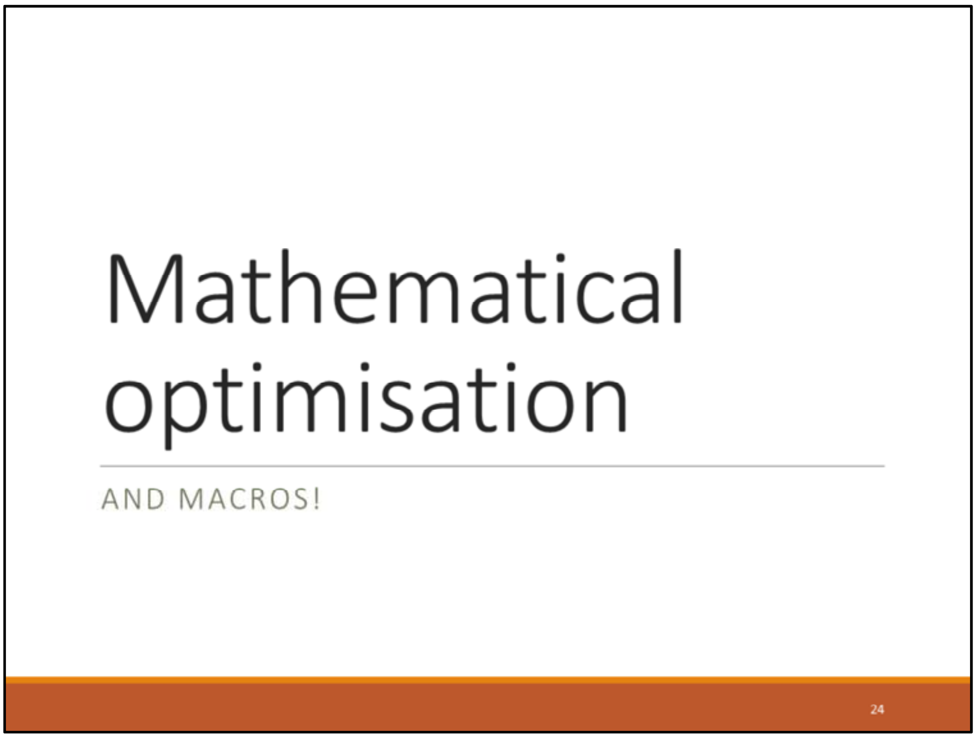
- Run the Web server:

```
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”)
```



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

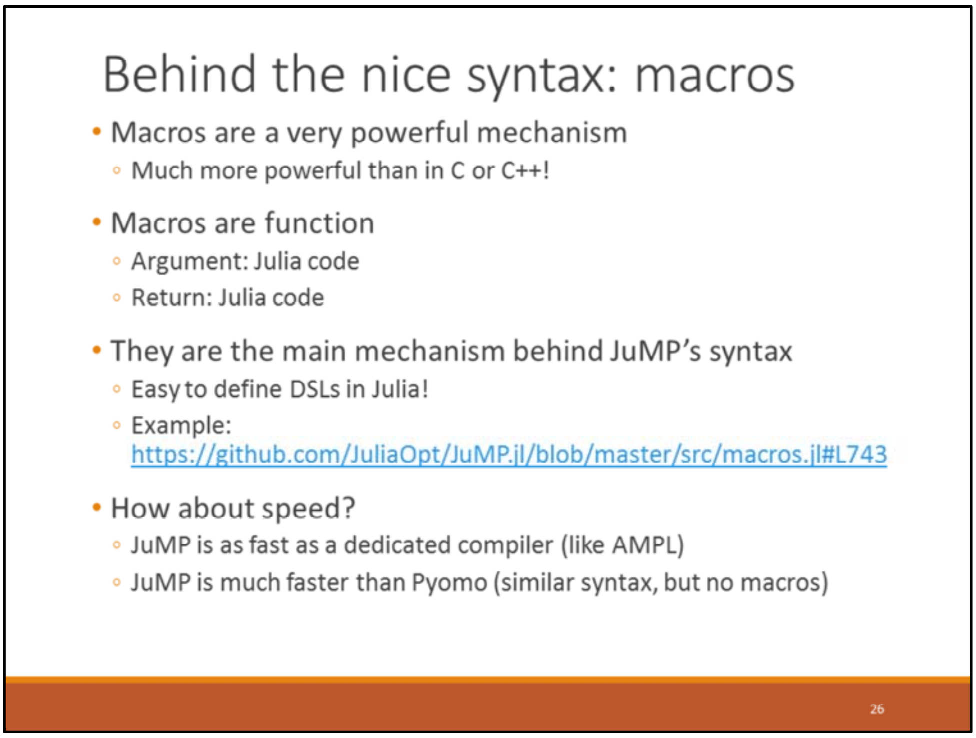
```
julia> Pkg.add(”JuMP”)
julia> Pkg.add(”Cbc”)
julia> using JuMP
```

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

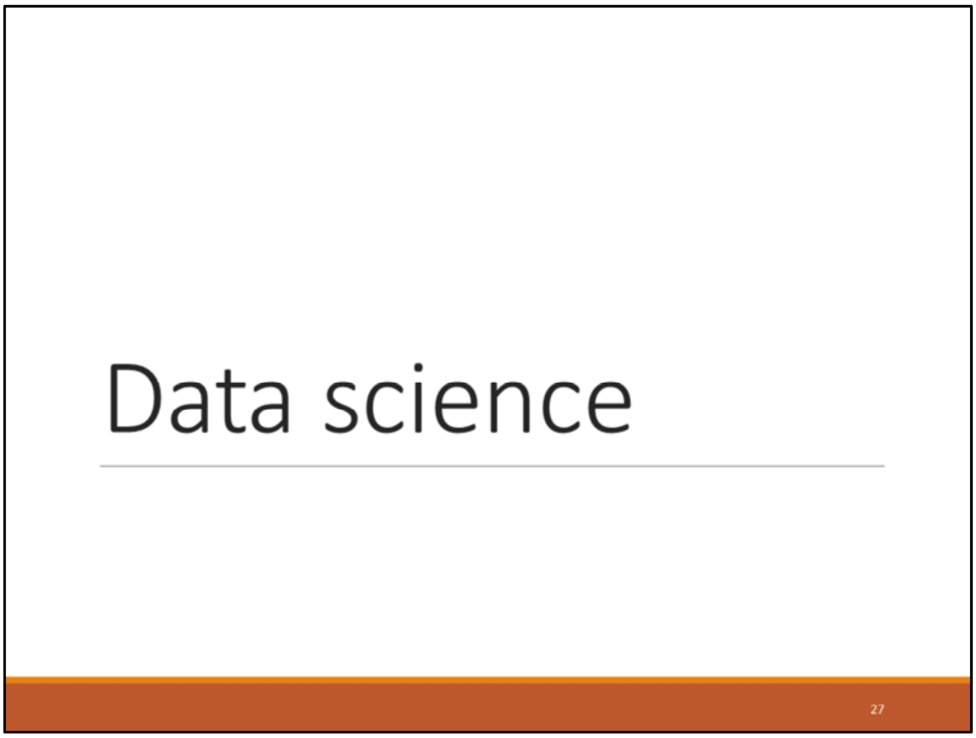
```
m = Model()
@variable(m, x >= 0)
@variable(m, 1 <= y <= 20)
@objective(m, Max, x + y)
@constraint(m, 2 * x + y <= 8)
solve(m)
```

25

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.



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.



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

Data frames: **DataFrames.jl**

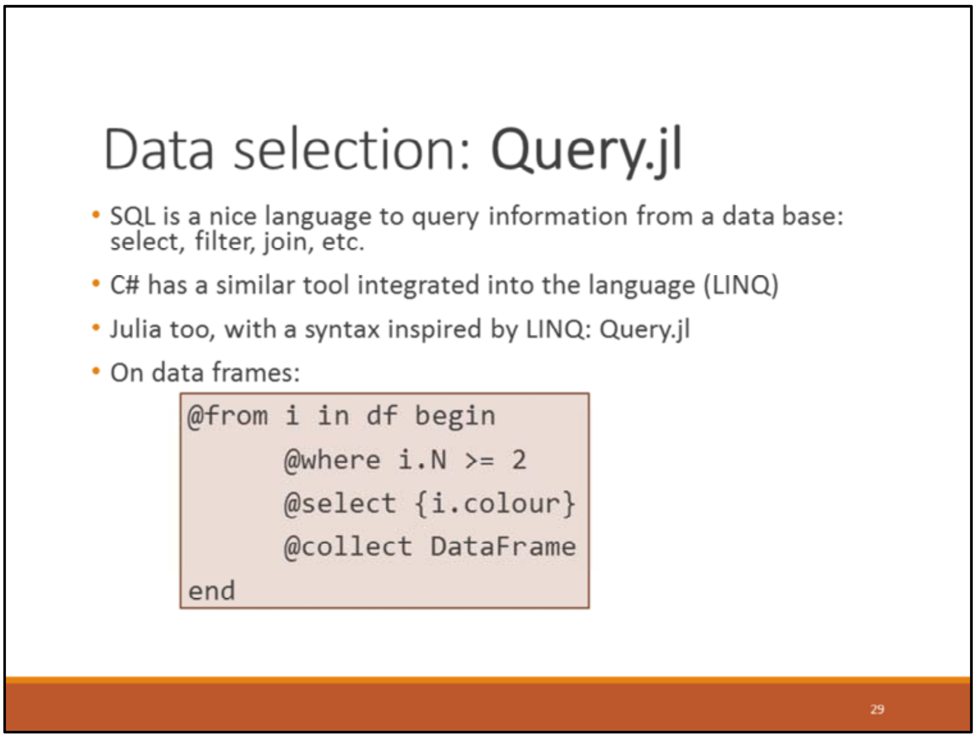
- R has the data frame type: an array with named columns

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

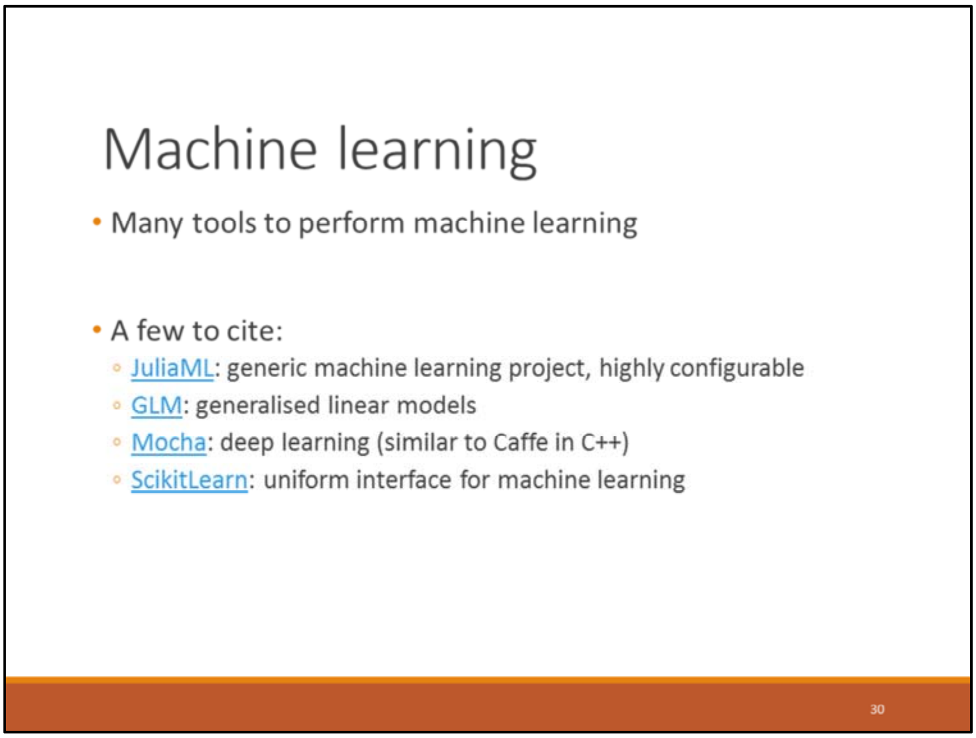
- Easy to retrieve information in each dimension:

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

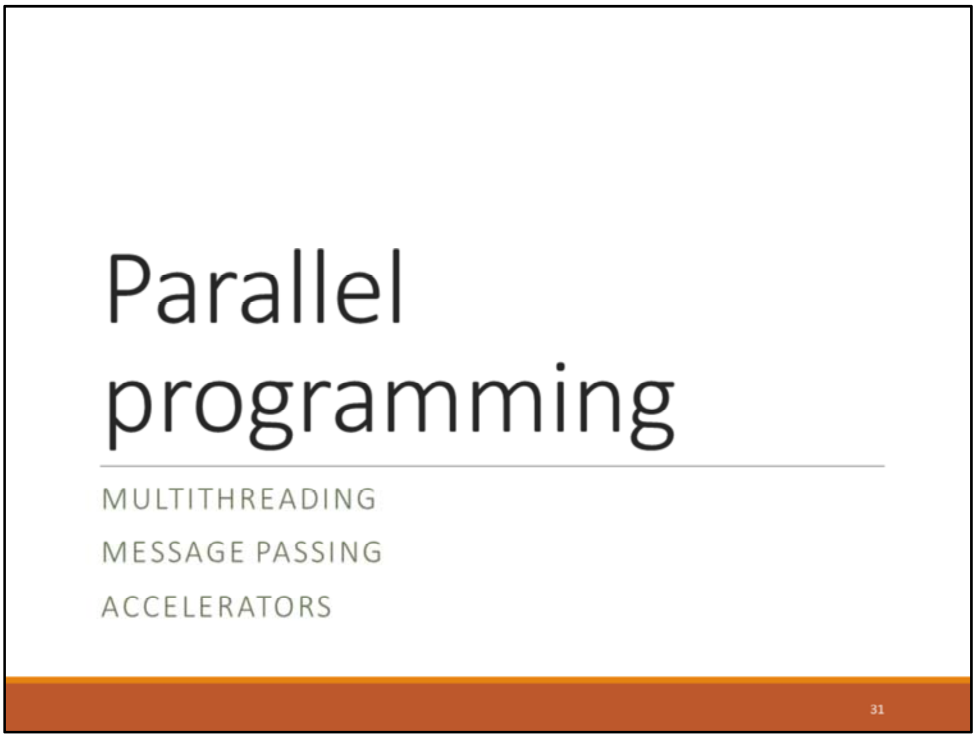
- The package has good support in the ecosystem
  - Easy plot with Plots.jl: just install StatPlots.jl, it just works
  - Understood by machine learning packages, etc.



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



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

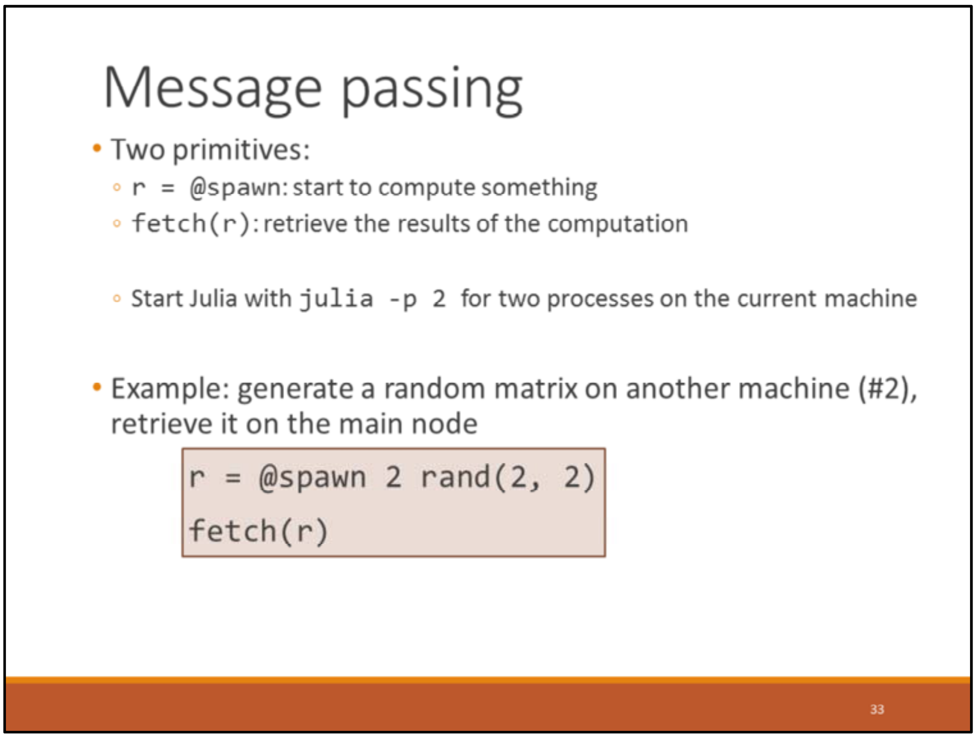


For scientific computations… and big data (to name a few), parallel is needed: must solve very large problems, deal with enormous quantities of data.
Multiple paradigms so far: multithreading (cores of a machine), message passing (multiple machines), accelerators (GPUs). All three are currently supported 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



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

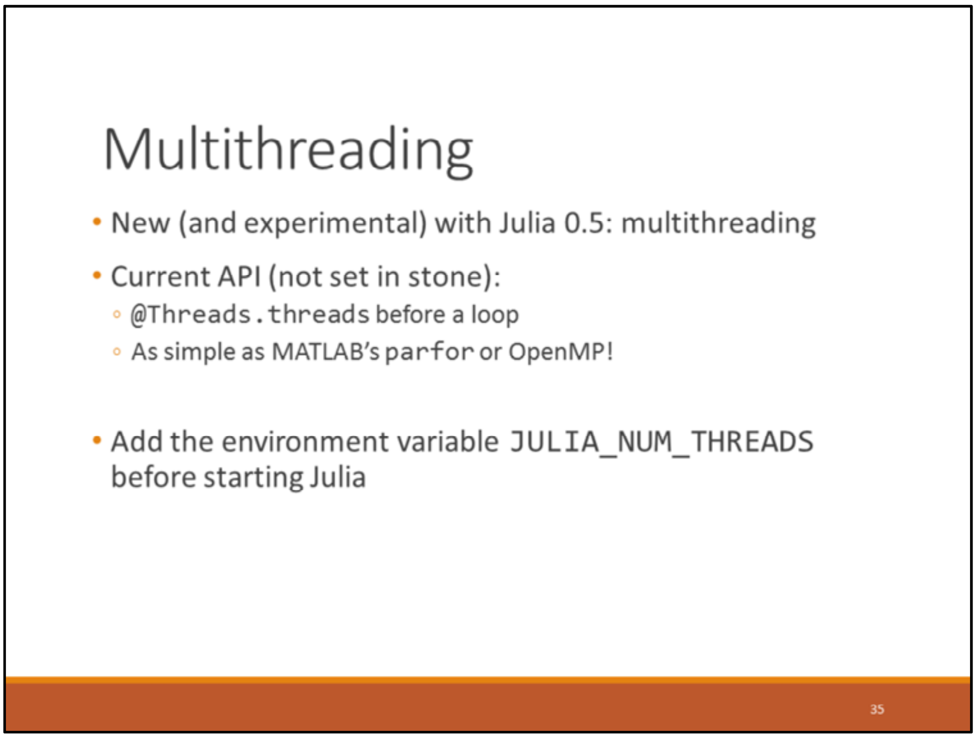
```
#!/bin/bash
#SBATCH --job-name="juliaTest"
#SBATCH --output="juliaTest.%j.%N.out"
#SBATCH --partition=compute
#SBATCH --nodes=8
#SBATCH --export=ALL
#SBATCH --ntasks-per-node=24
#SBATCH -t 01:00:00
export SLURM_NODEFILE=`generate_pbs_nodefile`
./julia --machinefile $SLURM_NODEFILE test.jl
```

Message passing: reductions

- Hadoop uses the map-reduce paradigm
- Julia has it too!

- Example: flip a coin multiple times and count heads

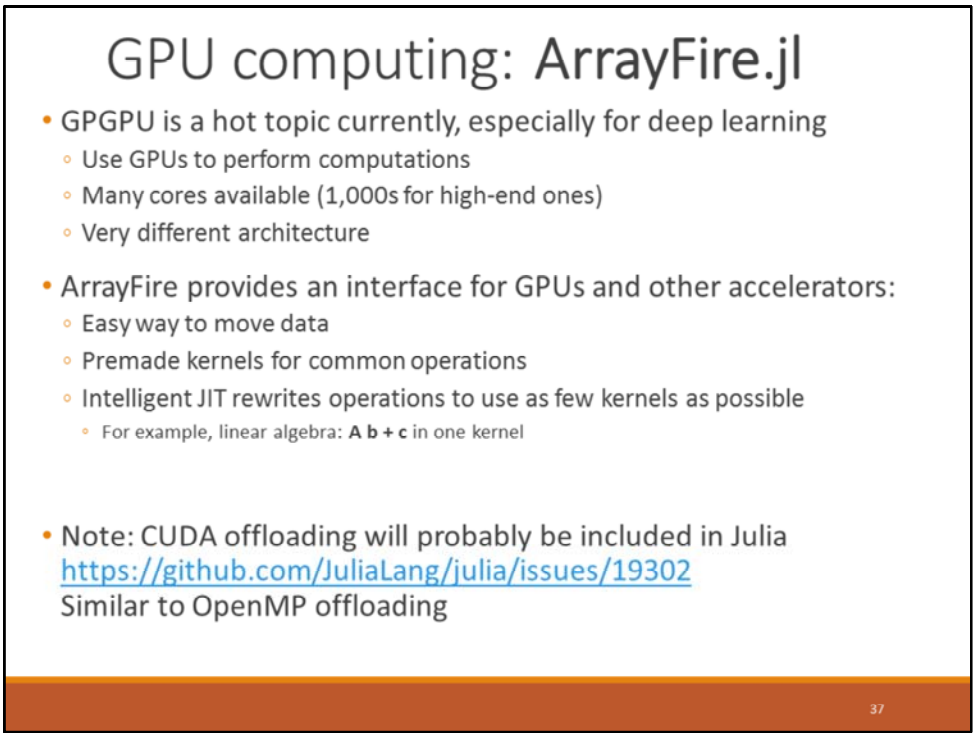
```
nheads = @parallel (+) for i in 1:500
        Int(rand(Bool))
end
```



Should be finalised with Julia 1.0.

Multithreading

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



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:

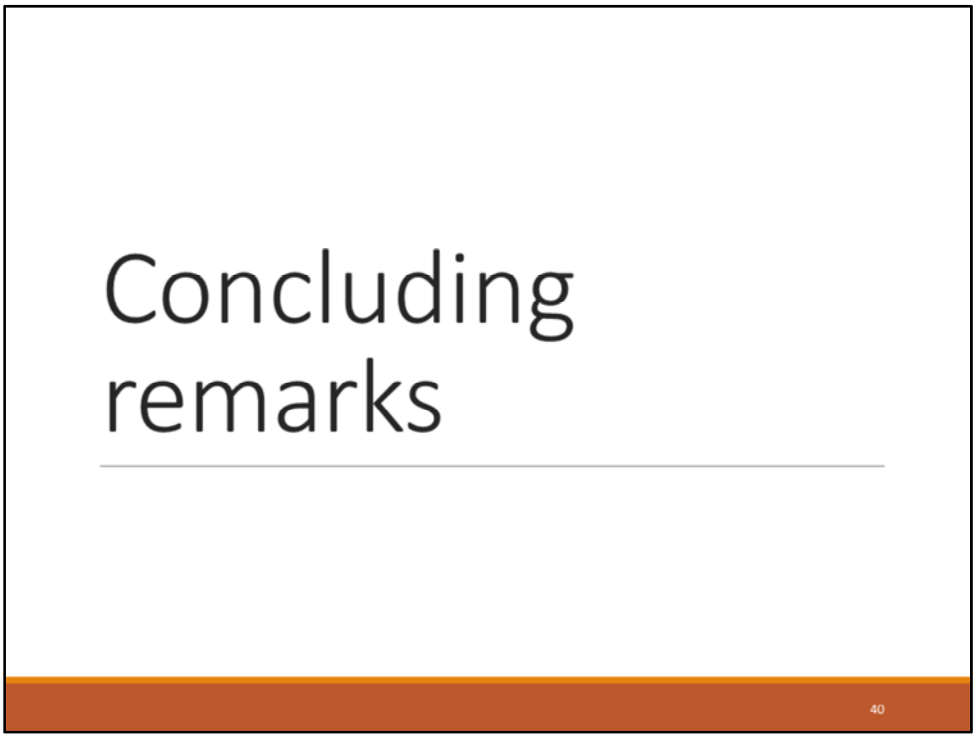
```
a_cpu = rand(Float32, 10, 10);
a_gpu = AFArray(a_cpu);
b_gpu = AFArray(rand(Float32, 10, 10));
```

- Then work on it as any Julia array:

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

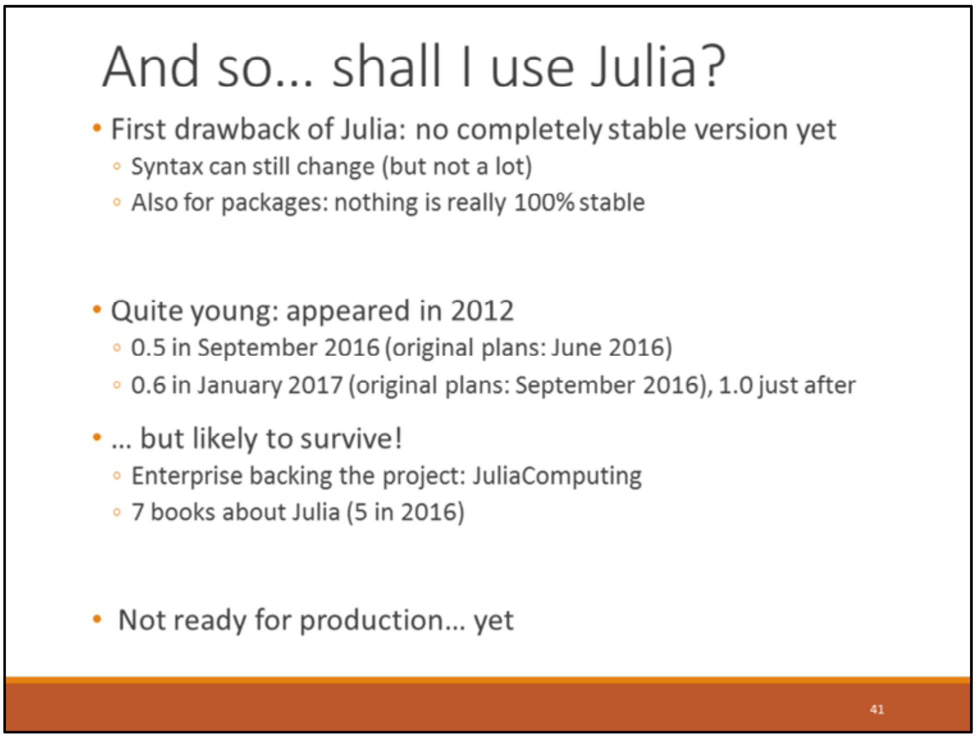
- Finally, retrieve the results:

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



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?



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