Scientific Programming in Julia

Plots and High Performance Computing

Nicholas Gale and Stephen Eglen

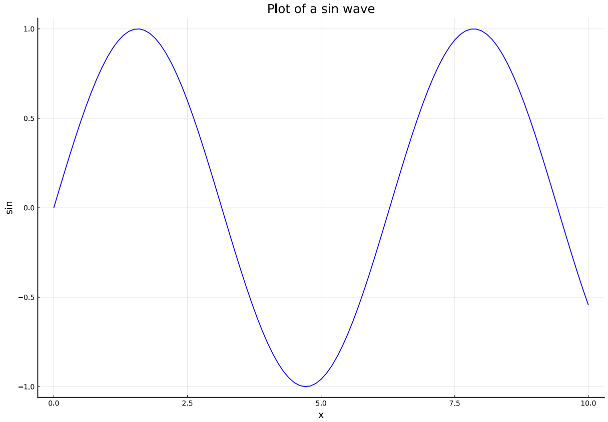
Plots

- Plotting is typically done through the `Plots` package.
- Plotting in Julia is relatively immature: it requires a backend.
- Select one of PyPlot, Plotly, GR, … (I normally go for GR for speed+features).
- There is a high time-to-first plot time but it is smooth from there.

Plotting grammar

- Plotting follows a predictable grammar through the `plot` API call.
- Plots are arranged in x data, y data formats. Series are delineated with spaces.
- Optional keywords denote styling and can be shorted:
  1) seriestype/st = {“scatter”, “line”, “contour”, etc.}
  2) color/c = {:red, RGB(0,1,0), etc} (can be input as a list)
  3) label = “series label”
  4) xlabel= , title= , ylabel=

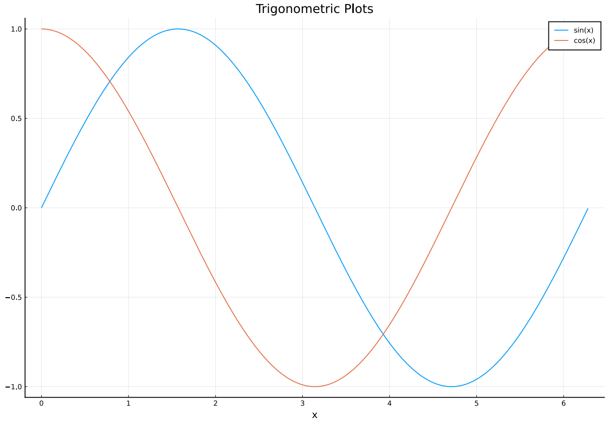
```
using Plots
pyplot()  # need pyplot for quarto notes
x = collect(0:0.1:10)
y = sin.(x)
plot(x, y; st="line", label=false, title="Plot of a sin wave",
     ylabel="sin", xlabel="x", colour=:blue)
```



Plot Objects

- Plots can be attached to objects; last plot is attached to an internal object.
- Plot objects can be added to with the ‘plot!(plot_object, x,y...; keywords=...) call.
- Plot objects can be referenced later.

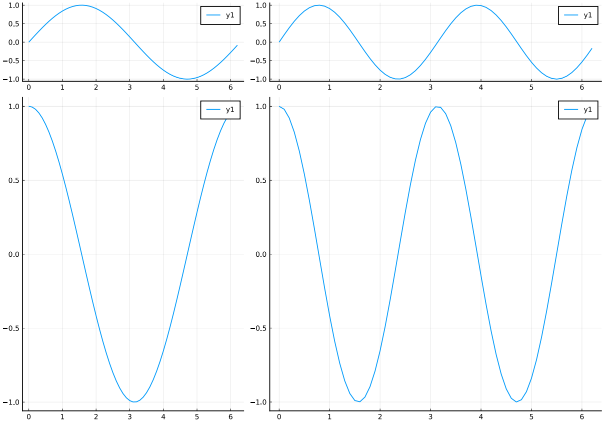
```
x = 0:0.01:2pi
y1 = sin.(x); y2 = cos.(x)

trig_plot = plot(x, y1; title="Trigonometric Plots",
                  xlabel="x", label="sin(x)")
plot!(trig_plot, x, y2; label="cos(x)")
```



Plot Layouts

- Subplots are handled through layout specification.
- Simple layouts can specify a grid format. A more complicated layout can be defined with the `@layout`
- A layout is arranged as a matrix with `a,b,c` denoting the plots. They are optionally given widths and heights e.g. `a{0.2h 0.3w}`.

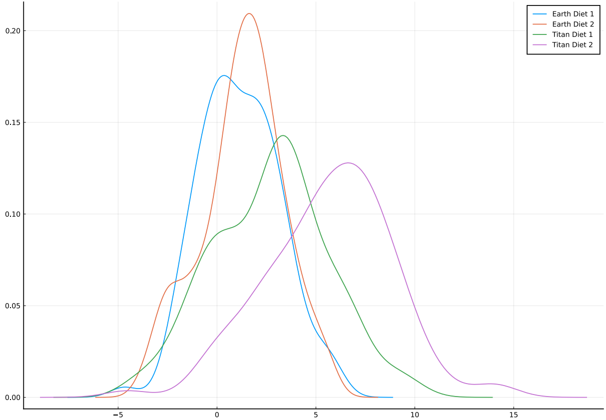
```
x = 0:0.1:2pi
p1 = plot(x, sin.(x)); p2 = plot(x, sin.(2x));
p3 = plot(x, cos.(x)); p4 = plot(x, cos.(2x));
L = @layout [a{0.2h, 0.4w} b; c d]
plot(p1, p2, p3, p4; layout=L)
```



StatsPlots

- Plots supports recipes and thus a Plotting ecosystem.
- StatsPlots allows for efficient DataFrames plotting.
- A slightly more complex grammar; documentation found here

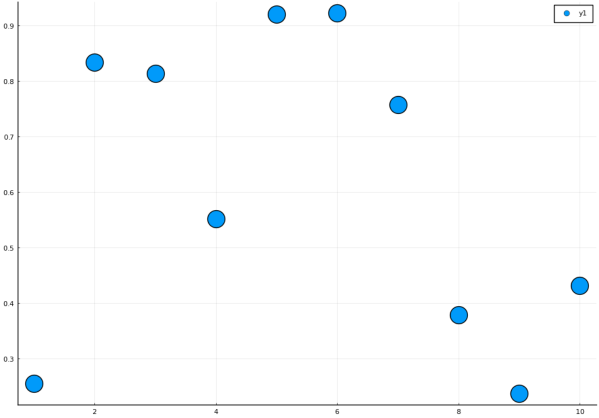
```julia
using DataFrames, CSV, StatsPlots

experiment = CSV.File("plotting_data.csv") ▷ DataFrame
@df experiment density(:BodyLength, group = (:Planet, :Diet),
                        legend=:topright)
```



Save Plots

- Plots are saved through the `savefig` API call.
- It specifies the path and plot object to be saved.
- The path defines the file type.

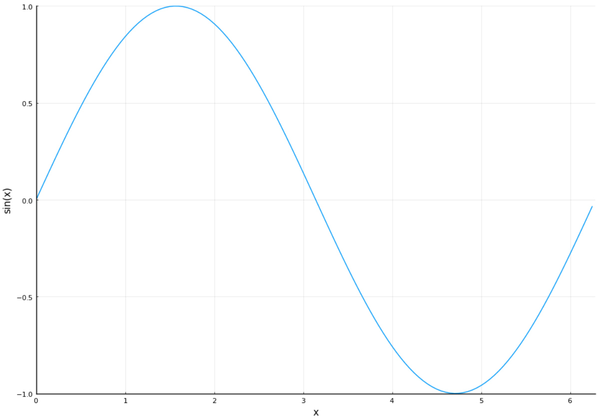
```
plt = plot(collect(1:10), rand(10); st=:scatter,
          markersize=20)
savefig(plt, "my_first_plot.png");
```



Animate

- A useful tool to visualise data is animation.
- This is done through the `@animate` macro applied to a `for` loop.
- The animation can be saved through the `gif` API call. This takes the animation object, then path, and optional FPS.

```
x = collect(0:0.05:2pi)
y = sin.(x)
anim_obj = @animate for t in 1:length(x)
    plot(x[1:t], y[1:t]; xrange=(0,2pi), yrange=(-1,+1),
          legend=false, xlabel="x", ylabel="sin(x)")
end
gif(anim_obj, "my_first_animation.gif", fps=30);
```



Performance Computing

- Efficient and fast code is one of the **big** draw cards of Julia.
- It *usually* comes for free, but not always.
- There are some common ‘tricks’ to employ to improve code efficiency and speed.

Data Access Patterns

- Data in Julia is organised in a *column-major* format.
- Data is laid out with the columns stacked end-to-end. The fastest way to access them is through the *rows*.
- Slow code uses columns as the 'fast-changing' index, but is more traditional. When looping use columns as the outer variable.

```julia
using BenchmarkTools

l = 10^4
A = rand(l,l);

sum1 = function(A, l)
    sum = 0.0
    for i = 1:l, j=1:l
        sum = sum + A[i,j]
    end
    sum
end

s = sum(A); s1 = sum1(A, l)
(s ≈ s1) || error("s1 wrong")
@btime sum1(A, l);
```

237.836 ms (1 allocation: 16 bytes)

```julia
# .
# .
# .
# .
# .
sum2 = function(A, l)
    sum = 0.0
    for j = 1:l, i=1:l
        sum = sum + A[i,j]
    end
    sum
end

s2 = sum2(A, l)
(s ≈ s2) || error("s2 wrong")
@btime sum2(A, l);
```

93.015 ms (1 allocation: 16 bytes

@inbounds

- Bounds checking is a useful operation in an interpreted language throwing an error if accessing an invalid index.
- It is expensive and fast languages like C don't use it - at user peril!
- It is activated with the `@inbounds` macro and distributes to all nested loops.
- Can also use command-line switch '–check-bounds={yes|no|auto*}'.

```julia
x = [5, 6, 7]
[ x[i] for i in 1:3]        # okay

## [ x[i] for i in 1:4]   # bounds error
##[ @inbounds x[i] for i in 1:4] # ?? generates error in repl

function check()
    x = [5, 6, 7]
    [@inbounds x[i] for i in 1:4]
end
check()
```

```
4-element Vector{Int64}:
          5
          6
          7
 4323828915
```

@SIMD

- SIMD stands for single instruction multiple data and is a machine level optimisation in modern CPUs.
- It allows many mathematical operations to be vectorised and optimised within the CPU cycle.
- Turn it on using the ‘`@simd` macro. For well organised data you can expect some speed up.
- See extended example `timeit.jl` in repository. On my Macbook pro, it went from 2.3 GFlop/s to 18.1 GFlop/s with @inbounds and @simd in tandem.

Type mutation

- Julia has excellent typing and type inference but abstraction puts demands on the compiler.
- When types mutate the compiler works hard to “deal with it” which is good for the end user.
- To avoid this Julia can strictly type variables and functions which alleviates the pressure.

```julia
a::Array{Float32, 1} = [4.0, 2.0, 1.2] # a strictly typed vector
f(x::Array{Float32, 1}) = sum(x) # a strictly typed function.
```

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

High Performance Computing

- HPC refers to distributing compute tasks in an efficient way.
- It typically refers to parallel computing which comes in two general flavours: mutliple CPU threads, or GPUs.
- These follow similar principles but the architectures require different coding styles.
- Julia abstracts many of the ‘gotchas’.

Distributed

- Parallel computing with multiple threads is available through the `Distributed` package.
- It supports low level (`spawn`, `fetch`, `remotecall` etc.) methods.
- More often it is used to parallelise loops through the `@distributed` macro before a `for` loop block.
- To make processes available use the `addprocs(n)` method.

```julia
using Distributed
addprocs(4)

@time for i = 1:10^9
    i^2
end

@time @sync @distributed for i = 1:10^9
    i^2
end
```

```
  0.000000 seconds
 1.063840 seconds (582.35 k allocations: 38.901 MiB, 27.94% compil
```

```
Task (done) @0x000000010a576400
```

Distributed Reduce

- A useful operation for parallesiation is reduction.
- A distributed for block can specify a binary operation to reduce on.

```julia
sumodds = @distributed (+) for i = 1:100
    Int(isodd(i) && i)
end
```

```
2500
```

Shared Functions

- Each thread has access to its own local environment and thus function definitions.
- The `@everywhere` macro is used to specify that a function/package can be accessed from thread.

```
using Distributed
@everywhere function myfunc(x)
    return x^2 - y^3 + sin(x * y)
end

@everywhere using Statistics
```

Shared Data

- Additional threads also do not have access to data on the master thread.
- The `SharedArrays` package allows for memory to be shared between threads through a `SharedArray` object.

```
using Distributed
@everywhere using SharedArrays

v = SharedArray(zeros(5))

@sync @distributed for i = 1:length(v)
    v[i] = i^2
end
println(v)
```

```
[1.0, 4.0, 9.0, 16.0, 25.0]
```

Race Conditions and Synchronisation

- Data and thread access is typically asynchronous.
- What happens when two threads depend on each other catastrophically?
- What happens when two threads try to access/modify data at the same time.
- These are *race* conditions and the second can be dealt with using *atomic* operations and synchronisation.

pmap

- Functional progamming styles are supported through pmap which behaves like map.
- There is a slight performance cost and is best used with complex function calls.

```
pmap( x -> x^2, 1:5)
```

```
5-element Vector{Int64}:
  1
  4
  9
 16
 25
```

GPU Compute

- Graphical Processing Units use a special hardware layout to launch hundreds of threads with a low clock speed.
- They are slow at performing individual tasks, but can do many simultaneously.
- The startup time is slow but for large jobs speed ups of 20x to 100x are commonplace.
- Julia supports them through a generic `GPUArrays` backend, but users interface with a card specific API: `AMDGPUs`, `CUDA`, `Metal`.
- They are best used for linear algebra and vectorisable tasks.

GPU Basics

- APIs have a shared array data structure which can load CPUs to the GPU: `CUDA.CuArray`, `AMDGPU.ROCArray`, `Metal.MtlArray`.
- Common functions are overloaded to these data structures: `sum`, `+`, `-`, `*`, `/`, `^`, `.`, `sin`, `cos`, `exp` etc.
- Functional programming is supported through `map` and `reduce` frameworks.
- Indexed functions are *highly* discouraged: the GPU has to fall back on the CPU incurring a *very* high cost.
- Calls are asyncronous and the `PACKAGE.@sync` macro will synchronise data which is important for data dependencies.

```julia
using CUDA
a = rand(10^8);
a_gpu = CuArray(a);
@time sum(a);
CUDA.@time CUDA.@sync sum(a_gpu)
```

Further reading

- Performance tips:
  https://docs.julialang.org/en/v1/manual/performance-tips/
- Static arrays can often be faster (for small arrays up to about 100 elements):
  https://github.com/JuliaArrays/StaticArrays.jl
- Floops.jl for more advanced speed ups – see Alan Edelman's birthday problem video:
  https://www.youtube.com/watch?v=dczkYIOM2sg
- Julia style guide:
  https://docs.julialang.org/en/v1/manual/style-guide/

Summary

- Plotting
- Performance
- High-performance computing