What Is Julia?

Julia is a modern, open-source high-level programming language designed scientific computing. Though Julia was developed with this focus in mind, it remains a general-purpose language (closer to Python than R or Matlab) that can be used to solve a wide range of programming problems.

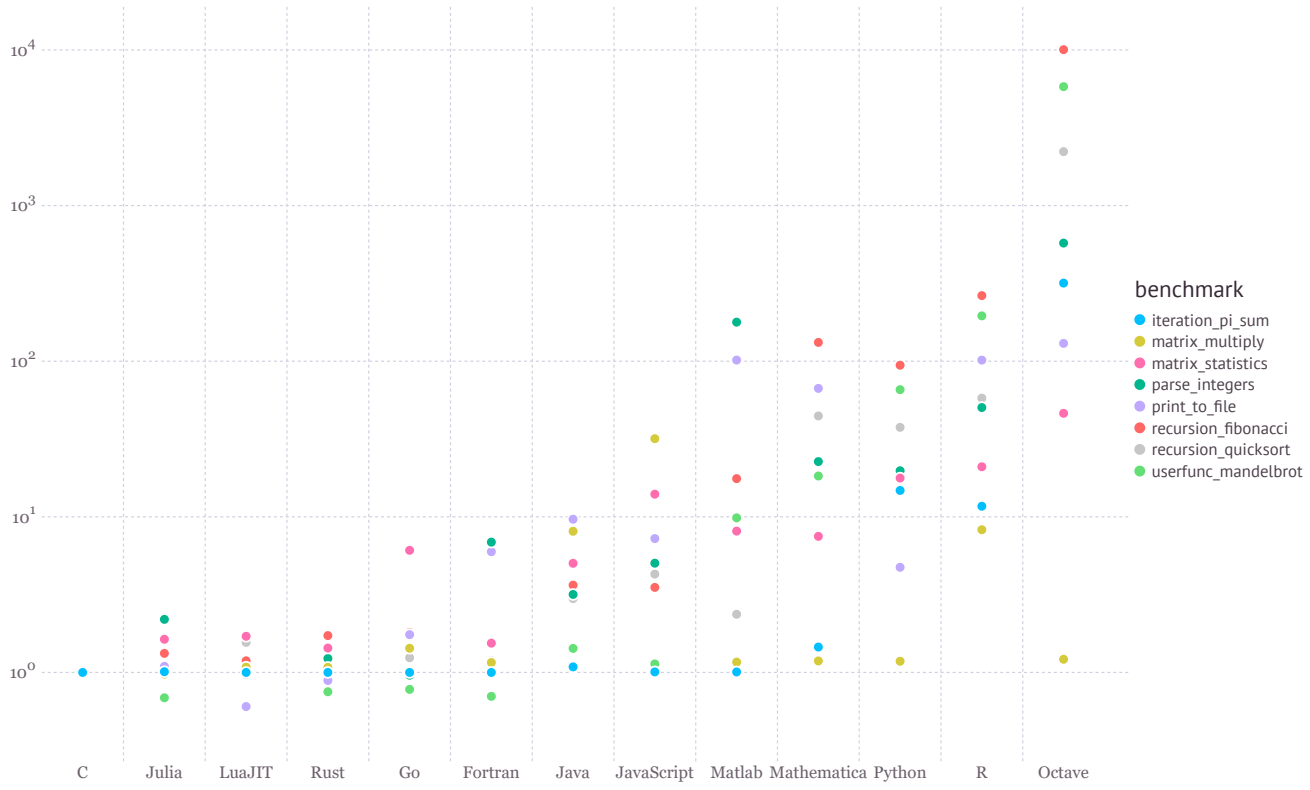
Why Use Julia?

The Two-Language Problem

High level languages like Python have become quite popular in scientific computing as their ease-of-use allows scientists to quickly prototype new programs and easily adapt them to changing requirements. Unfortunately, this comes at a bit of a cost as programs written in Python often struggle with performance. This means that, as the volume of data needing to be processed grows, Python programs must often be re-written a lower level language (like C/C++) to achieve an acceptable level of performance.

Once converted to a lower level language, however, updates to the program or the addition of new features becomes more difficult — new features may need to be prototyped again in Python before a *second rewrite* into a lower level language. Julia is positioned to solve this problem of continuous code re-writing and translation by providing a high-level language for developers, but also a high performance compiler capable of generating the optimised programs needed in scientific computing.

Just How Fast?



How Does Julia Compare to X?

The Julia documentation contains several pages of information explaining the important differences between Julia and other popular languages like Python, R, MATLAB, C/C++, and Common Lisp.

To summarise where Julia draws it's inspiration from and how it improves on these other languages:

- Julia, like MATLAB and R, is a programming language developed with scientific computing in mind, but (unlike MATLAB) is open-source and (unlike R) is flexible enough to still allow for comfortable general-purpose programming (as in Python)
- Julia combines the high-level programming and easy syntax of Python, with the speed and efficiency of C/C++
- Julia tops this all off with powerful features like multiple dispatch and metaprogramming borrowed from Common Lisp, but without the comparatively arcane syntax

Why *Not* Use Julia?

Though Julia is an excellent programming language well-posed to solve the two-language problem for most people, it's not for everyone. There remain a number of domains in which Julia as a language struggles:

- **Responsiveness ::** Julia's JIT compiler that manages to deliver such high-performance code in a dynamic language can also lead to large latency when new code is being run for the first time. It can still take tens of seconds to load a new package or even longer to plot some basic data. Julia isn't a language for embedded or realtime applications
- **Julia programs are for Julia users ::** Julia binaries are large (up to GBs) and difficult to share on their own. Source code is more commonly shared than compiled programs, so every user of your code will need their own Julia installation — something that isn't significantly different from other interpreted languages like MATLAB, R, and Python, but that is from C/C++. Julia is, generally speaking, a larger installation than R or Python, but still smaller than something like MATLAB
- **A relatively young language ::** Julia is still relatively new and there are some rough edges still. The ecosystem around the language is still changing quickly and up-to-date learning resources are comparitively hard to come by (particuarly when compared to a ubiquitious lanugage like Python)
- **And more...**

A Language for Science

Interativity

The Julia REPL

Julia comes with a richly-featured Read-Evaluate-Print-Loop (REPL) that can be used for interactive development. The REPL comes with several modes and distinctive features:

- Support for adding Unicode characters using LaTeX (try `\pi <tab>`)
- In-built package managment with Pkg mode (no `pip` needed!)
- Access to a Unix shell in Shell mode (don't need a second terminal window)
- A help mode for accessing documentation and code examples

Pluto.jl

Pluto is an interactive notebook environment (like Jupyter notebook), but is reactive (cells are automaticaly re-run when their inputs change) and is written in and optimised for Julia.

This notebook is written in Pluto.jl and you can play with Pluto yourself without installing on Binder. The Pluto examples showcase many more advanced features than are used here!

Matematical Notation

Unicode Support

```
2.5231325220201604
```

```
# Get the radius of a circle with an area of 20
begin
    A = 20
    √(A / π)
end
```

Literal Coefficients

```
x = 3
```

```
x = 3
```

```
10
```

```
# Clean polynomial functions
2x^2 - 3x + 1
```

```
64
```

```
# And exponential ones too
2^2x
```

Imaginary & Rational Numbers

```
-1 + 0im
```

```
# Built-in support for imaginary numbers with `im`
im^2
```

```
2 - 2im
```

```
2(1 - 1im)
```

```
2//3
```

```
# And rational numbers with `//`
6//9
```

```
-1//2
```

```
-4//8
```

```
-1//3
```

```
5//-15
```

```
1//3
```

```
-4//-12
```

```
5//8
```

```
3//4 * 5//6 # 15//24
```

Working With Matrices

```
Hm = 2×2 Matrix{Int64}:
      1   1
      1  -1
```

```
# Support for matrix literals
Hm = [1 1; 1 -1]
```

```
H = 2×2 Matrix{Float64}:
     0.707107   0.707107
     0.707107  -0.707107
```

```
# And operations
H = 1/√2 * Hm
```

```
plus = 2×1 Matrix{Float64}:
        0.7071067811865475
        0.7071067811865475
```

```
# Including matrix multiplication
plus = 1/√2 * [1 1]'
```

```
2×1 Matrix{Float64}:
  1.0
 -0.0
```

```
round.(H * plus)
```

```
▸ [1.0, 1.41421, 1.73205, 2.0]
```

```
# Note the use of the `.` for broadcasting functions over a tensor
sqrt.(1:4)
```

Scientific Packages

The Julia ecosystem contains a robust set of packages for scientific computing. Today we'll be using some of the packages in StatsKit — particularly Dataframes.jl and GLM.jl.

Growth Curve Analysis in Julia

```
# Import some helpful packages for loading and plotting data
using CSV ✓, Dates ✓, DataFrames ✓, Gadfly ✓, GLM ✓, Statistics ✓
```

```
df =
```

	Time	1	2	3	4	5	6	7
1	00:00:00	0.025	0.031	0.042	0.038	0.042	0.024	0.025
2	00:20:00	0.026	0.038	0.044	0.04	0.043	0.032	0.029
3	00:40:00	0.027	0.033	0.042	0.035	0.05	0.032	0.028
4	01:00:00	0.028	0.03	0.045	0.03	0.062	0.036	0.027
5	01:20:00	0.028	0.029	0.042	0.03	0.079	0.039	0.028
6	01:40:00	0.028	0.03	0.041	0.03	0.092	0.044	0.029
7	02:00:00	0.029	0.03	0.042	0.029	0.104	0.052	0.028
8	02:20:00	0.028	0.03	0.04	0.031	0.113	0.063	0.029
9	02:40:00	missing	0.03	0.04	0.031	0.119	0.067	0.03
10	03:00:00	missing	0.03	0.04	0.03	0.128	0.077	0.03
11	03:20:00	missing	0.031	0.041	0.029	0.125	0.089	0.031
12	03:40:00	missing	0.031	0.04	0.029	0.125	0.099	0.028
13	04:00:00	missing	0.031	0.041	0.029	missing	missing	missing
14	04:20:00	missing	missing	missing	missing	missing	missing	missing

```
# Load CSV into a DataFrame
df = DataFrame(CSV.File("Growth Curve Data.csv"))
```

ldf =

	Time	Replicate	OD
1	00:00:00	"1"	0.025
2	00:20:00	"1"	0.026
3	00:40:00	"1"	0.027
4	01:00:00	"1"	0.028
5	01:20:00	"1"	0.028
6	01:40:00	"1"	0.028
7	02:00:00	"1"	0.029
8	02:20:00	"1"	0.028
9	00:00:00	"2"	0.031
10	00:20:00	"2"	0.038
⋮	more		
299	02:20:00	"25"	0.82

```
# Convert to long format
ldf = stack(df, 2:26, variable_name=:Replicate, value_name=:OD) |> dropmissing
```

```
start = 00:00:00
```

```
# Normalise times and convert to minutes
start = ldf[1, :Time]
```

pdf =

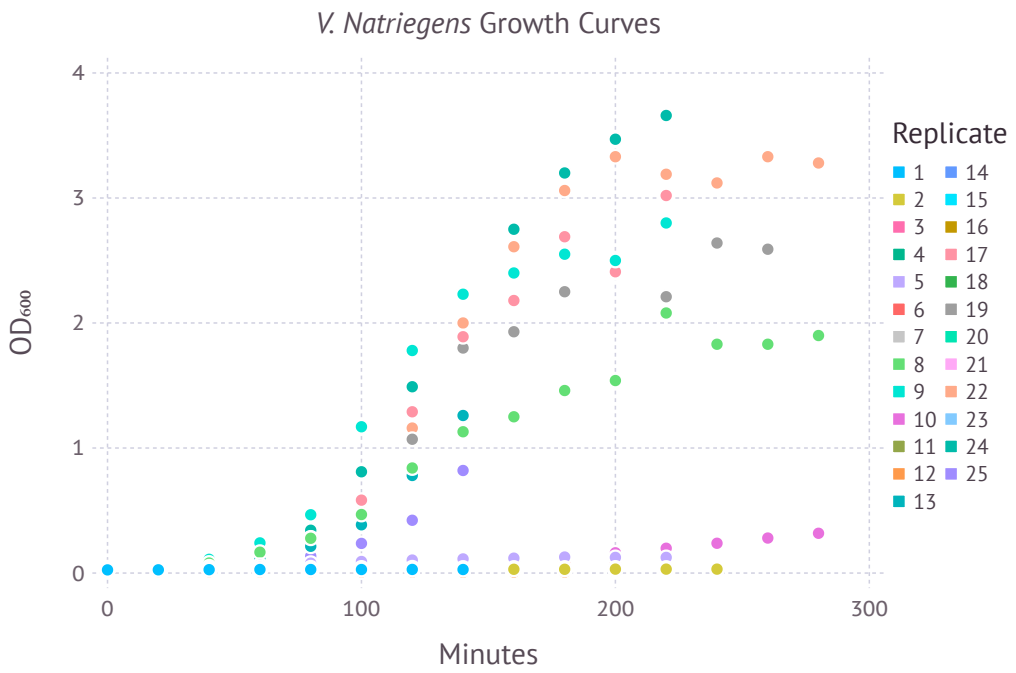
	Time	Replicate	OD
1	0	"1"	0.025
2	20	"1"	0.026
3	40	"1"	0.027
4	60	"1"	0.028
5	80	"1"	0.028
6	100	"1"	0.028
7	120	"1"	0.029
8	140	"1"	0.028
9	0	"2"	0.031
10	20	"2"	0.038
⋮	more		
299	140	"25"	0.82

```
pdf = transform(ldf, :Time => ByRow(t -> Dates.value(Minute(t - start))) => :Time)
```

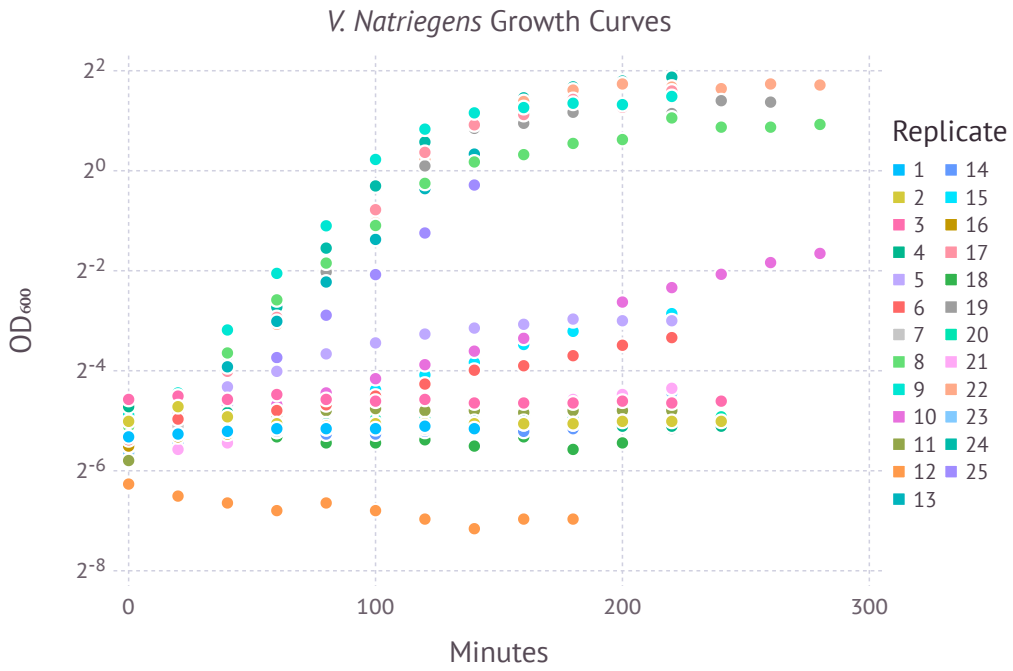
Growth Curve Data

Given that OD is proportional to cell count (when properly diluted so that readings don't exceed 0.6), it can be used to track the growth of cells.

On a linear scale, this growth curve is an exponential, but be later made linear by applying a logarithmic transformation.




```
# Construct a line-scatter plot, grouping by biological replicate
plot(pdf, x=:Time, y=:OD, color=:Replicate,
     Guide.xlabel("Minutes"), Guide.ylabel("OD₆₀₀"),
     Guide.title("<i>V. Natriegens</i> Growth Curves"))
```




```
# Replot, but on a log-scale so that we can pick out the exponential growth region
plot(pdf, x=:Time, y=:OD, color=:Replicate, Scale.y_log2,
     Guide.xlabel("Minutes"), Guide.ylabel("OD₆₀₀"),
     Guide.title("<i>V. Natriegens</i> Growth Curves"))
```

```
gdf =
```

GroupedDataFrame with 25 groups based on key: Replicate

First Group (8 rows): Replicate = "1"

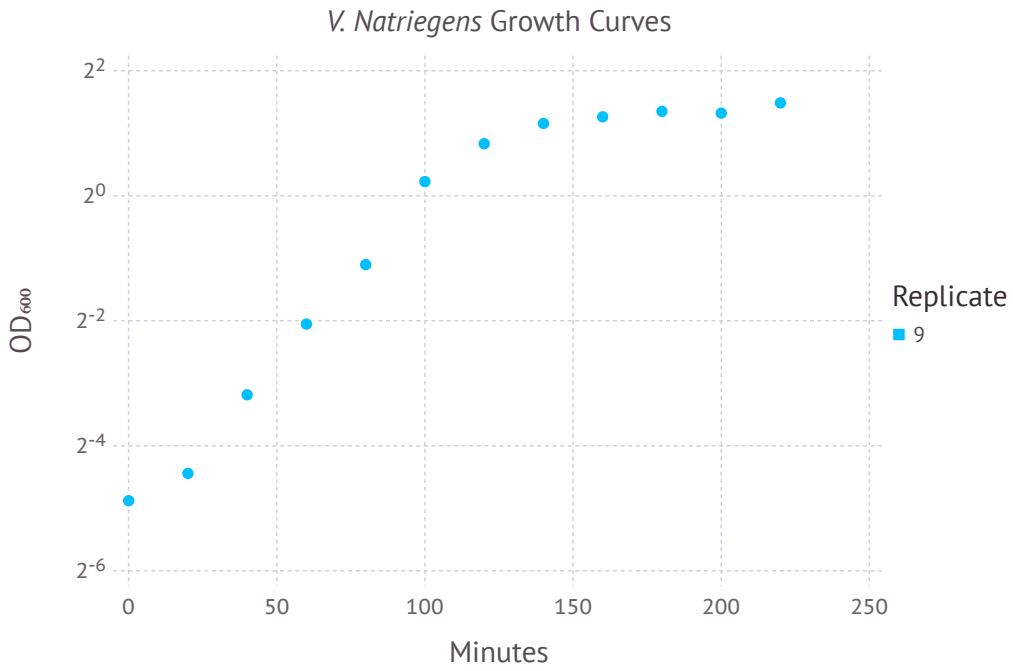
	Time Int64	Replicate String	OD Float64
1	0	1	0.025
2	20	1	0.026
3	40	1	0.027
4	60	1	0.028
5	80	1	0.028
6	100	1	0.028
7	120	1	0.029
8	140	1	0.028

⋮

Last Group (8 rows): Replicate = "25"

	Time Int64	Replicate String	OD Float64
1	0	25	0.028
2	20	25	0.025
3	40	25	0.039
4	60	25	0.075
5	80	25	0.135
6	100	25	0.237
7	120	25	0.422
8	140	25	0.82

```
# Group by replicate / media
gdf = groupby(pdf, :Replicate)
```




```
begin
    # Pick a replicate to zero in on
    curve = 9
    # Replot, but focusing on a single curve
    plot(gdf[curve], x=:Time, y=:OD, color=:Replicate, Scale.y_log2,
         Guide.xlabel("Minutes"), Guide.ylabel("OD₆₀₀"),
         Guide.title("<i>V. Natriegens</i> Growth Curves"))
end
```

`logdf =`

	Time	Replicate	OD
1	20	"9"	0.046
2	40	"9"	0.11
3	60	"9"	0.241
4	80	"9"	0.466
5	100	"9"	1.17

```
# Trim the data to take a closer look at log-phase
logdf = filter(:Time => t -> 20 <= t <= 100, gdf[curve])
```

	Time	Replicate	OD
1	20	"9"	-4.44222
2	40	"9"	-3.18442
3	60	"9"	-2.05289
4	80	"9"	-1.1016
5	100	"9"	0.226509

```
# Log-transform the OD data
transform!(logdf, :OD => ByRow(log2) => :OD)
```

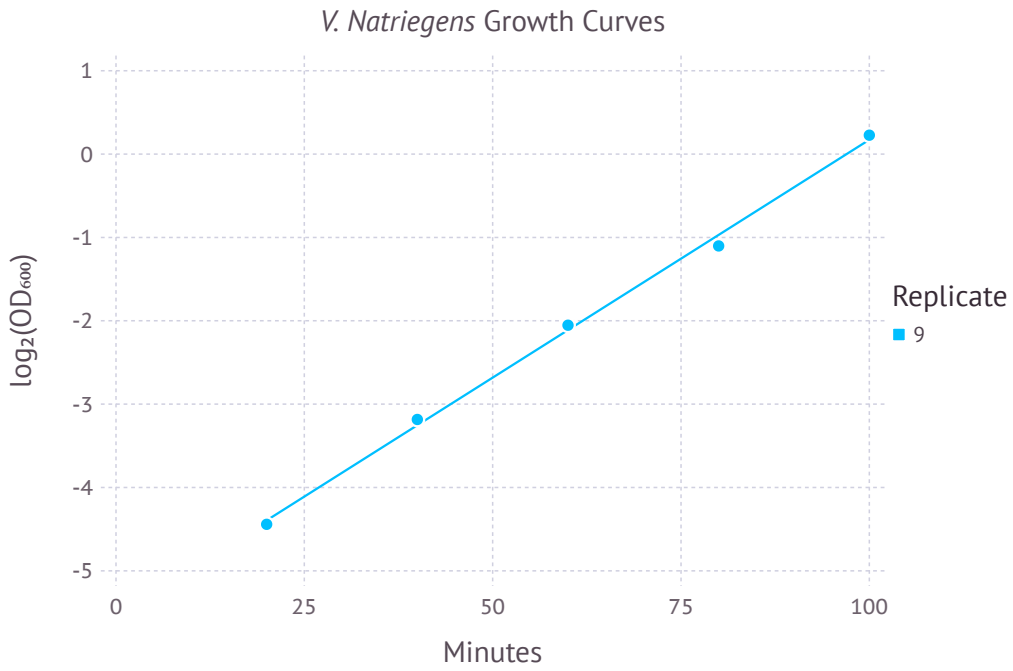
```
ols =
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredCho

OD ~ 1 + Time

Coefficients:
```

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	-5.53701	0.106191	-52.14	<1e-04	-5.87496	-5.19907
Time	0.0571014	0.00160089	35.67	<1e-04	0.0520067	0.0621962

```
# Perform and ordinary least-squares regression for a linear model
ols = lm(@formula(OD ~ Time), logdf)
```

```
[-4.39498, -3.25296, -2.11093, -0.968897, 0.173131]
```

```
# Insert a new column into our dataframe representing the model predictions
logdf[!,:Model] = predict(ols)
```




```
# And then plot it on a log-scale
plot(logdf, x=:Time, y=:OD, color=:Replicate, Geom.point,
     Guide.xlabel("Minutes"), Guide.ylabel("log₂(OD₆₀₀)"),
     Guide.title("<i>V. Natriegens</i> Growth Curves"),
     layer(x=:Time, y=:Model, Geom.line))
```

Calculating Doubling-Time From Our Model

We can start with a fundamental equation that models the growth of microbes undergoing binary fission:

$$N = N_0 2^{\frac{t}{g}}$$

Where N is the current number of cells, N_0 is the initial number of cells, t is time, and g is generation or doubling-time. We want to rearrange this equation to fit the model `OD ~ Time` after calculating the $\log_2$ of all ODs.

Let's start by applying the $\log_2$ to both sides of the equation:

$$\log_2 N = \log_2 N_0 + \frac{t}{g}$$

Ignoring the intercept and separating terms, we get an expression that matches our model:

$$\log_2 N = \frac{1}{g} t$$

Therefore we can conclude that g is equal to the reciprocal of our regression gradient.

The doubing time was ~17.5 minutes

```
begin
    # Calculate doubling-time
    g = 1/coef(ols)[2]
    # Format it into a nice string
    md"The doubing time was ~$(round(g, sigdigits=3)) minutes"
end
```