Julia for Engineering Students

Dr. Torsten Schenkel

Table of contents

Preface

This is a collection of notes I put together for myself and my students. It may grow into a book some day.

This "book" is an incomplete and strongly biased introduction to Julia for students and resesarchers in engineering disciplines. I put it together as a guide for my students and myself and publish it to help colleagues who have shown an interest in using Julia in their own research.

This book was created using Quarto, written on Emacs.

Code is active and the exercises and examples can be downloaded as Jupyter notebooks.

All content and examples are published as is and without any guarantee of usefulness or even fitness for purpose[1], let alone elegance.

[1] Although I do think that it will be both useful and is fit for purpose.

-

1 Introduction

1.1 What is Julia?

Julia is a programming language that is comparable to Matlab and Python, but also has some features of lower level languages like Fortran and C.

Julia is a dynamically typed, interactive language[2], developed and published by researchers at Massachusetts Institute of Technology (MIT)[3] and Julia Computing[4].

Julia is free and open-source software (FOSS), and is available for free for learning, research and commercial use. Commercial support is available from Julia Computing.

[2] It is used in a "read-eval-print-loop" (REPL).

[3] Funny enough that is the same place where Matlab was born.

[4] For details see Julia Governance.

1.2 How is Julia different?

1.2.1 The Julia Manifesto

When asked why they created Julia, the four founders Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman answered:

> In short, because we are greedy.
>
> We are power Matlab users. Some of us are Lisp hackers. Some are Pythonistas, others Rubyists, still others Perl hackers. There are those of us who used Mathematica before we could grow facial hair. There are those who still can't grow facial hair. We've generated more R plots than any sane person should. C is our desert island programming language.
>
> We love all of these languages; they are wonderful and powerful. For the work we do — scientific

computing, machine learning, data mining, large-scale linear algebra, distributed and parallel computing — each one is perfect for some aspects of the work and terrible for others. Each one is a trade-off.

We are greedy: we want more.

We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled.

(Did we mention it should be as fast as C?)

While we're being demanding, we want something that provides the distributed power of Hadoop — without the kilobytes of boilerplate Java and XML; without being forced to sift through gigabytes of log files on hundreds of machines to find our bugs. We want the power without the layers of impenetrable complexity. We want to write simple scalar loops that compile down to tight machine code using just the registers on a single CPU. We want to write A*B and launch a thousand computations on a thousand machines, calculating a vast matrix product together.

We never want to mention types when we don't feel like it. But when we need polymorphic functions, we want to use generic programming to write an algorithm just once and apply it to an infinite lattice of types; we want to use multiple dispatch to efficiently pick the best method for all of a function's arguments, from dozens of method definitions, providing common functionality across drastically different types. Despite

> all this power, we want the language to be simple and clean.
>
> All this doesn't seem like too much to ask for, does it?
>
> Even though we recognize that we are inexcusably greedy, we still want to have it all. About two and a half years ago, we set out to create the language of our greed. It's not complete, but it's time for an initial[1] release — the language we've created is called Julia. It already delivers on 90% of our ungracious demands, and now it needs the ungracious demands of others to shape it further. So, if you are also a greedy, unreasonable, demanding programmer, we want you to give it a try.

Ok, that's a lot of geekery in there, so what does that mean for you, the user of a scientific language?

2 Julia Features for Scientic Computing

While the question "What's the best programming language?" is impossible to answer[5], Julia has some specific features that make it particularly well suited for scientific computing:

[5] and the answer will be different for different applications.

- **just-in-time (JIT) compilation**, which allows performance that is on the same level as Fortran or C code.
- **multiple dispatch**, a programming paradigm, which, while a functional paradigm, brings some of the possibilities usually associated with object-oriented (OO) programming[6].
- **1-based indexing**[7], which makes the code more closely resemble much of the mathematical concept we want to express[8].
- **Matlab similar syntax**: coming from Matlab, Julia's syntax will be familar enough to allow a quick transition[9].

[6] Julia is not an OO language, which some may see as a disadvantage. Personally I never found OO to be useful for the kind of scientific programming I do.

[7] Julia supports arbitrary indexing, but 1-based is the standard.

[8] Some, coming from C or Python may see 1-based indexing as a negative.

[9] My first experience with Julia was, indeed, porting an ODE problem from Matlab to Julia. And most of the work was replacing two function calls and switching square for round brackets.

2.1 Just-In-Time Compilation and the *Two Language Problem*

Many scientific models are developed in *prototyping* languages like Python (or Matlab), since they are interactive, i.e., they run in an "read-eval-print-loop" (REPL), which offers immediate feedback and results, encouraging experimentation and "playing" with the code. This is a key feature for many scientists, as opposed to software developers, since it allows for quick development of a working model.

The problem with these interactive languages is, that they tend to be *interpreted*, and this much slower than *compiled* languages. We will see later that, e.g., Python can be up to 3 orders of magnitude slower than a compiled language, which also has huge implications on energy efficiency of scientific computing, as a contributor to climate change (Pereira et al. 2017, 2021).

Pereira, Rui, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. "Energy Efficiency Across Programming Languages: How Do Energy, Time, and Memory Relate?" In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, 256–67. SLE 2017. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3136014.3136031.

———. 2021. "Ranking Programming Languages by Energy Efficiency." *Science of Computer Programming* 205 (May): 102609. https://doi.org/10.1016/j.scico.2021.102609.

So with any interpreted language scientific computing will arrive at the point where the performance of the developed code is no longer sufficient. Usually the next step in these cases is either

a. abandonment of the model.
b. re-implementation of the model in a compiled language like Fortran or C.

This is what is called the **"Two Language Problem"**.

The two-language-problem is particularly concerning for scientific computing, since most engineers and natural scientists are not computer-scientists or software developers, or coders. Many of us do not readily speak a compiled language which would require writing code on a much lower abstraction level than most of us are comfortable with[10].

[10] I do know how to write Fortran, but it is a much more daunting prospect than doing the same thing in Julia (or Matlab or Python).

Julia can overcome this problem by virtue of being "just-in-time" compiled. This means that on the surface it looks like an interpreted language, while, behind the scenes and completely transparent to the user, the code is compiled to machine code and then executed. The Julia developers coined the phrase:

> "Feels like Python, runs like C."

With only little effort, we can write Julia code that runs at the same speed as optimised Fortran or C code[11].

[11] In fact, Julia is the only dynamically typed language that has managed to join the exclusive Petaflop Club of peak performance of greater than one petaflops (1015 floating point operations per second), scaling to over 1 million threads.

Part I

Julia Basics

I don't want to give a full introduction to Julia. There are other, better resources for that (see Appendix A).

3 Installing Julia

Julia can be installed on any computer, even if you do not have administrator rights.

3.1 Installation on Windows

For Windows users, Julia is available from the Microsoft App Store. Simply open the app store and search for Julia[12] and install it. This will use the Julia Updater to install Julia in the user's home folder, so no administrator rights are needed[13].

[12] Look for the distinctive three coloured dot icon.

[13] This makes this method of installation ideal for university computers.

3.2 Installation on MacOS and Linux

Go to the JuliaUp github page and follow the instructions there.

If you do not believe in reading the documentation of software you are going to install, this should do it[14]:

[14] No guarantees! If your computer breaks, don't blame me.

```
curl -fsSL https://install.julialang.org | sh
```

3.3 Starting the REPL

In Windows, start the REPL from the *Start* menu, in Linux, open a terminal and type:

```
julia
```

3.4 Installing packages

Julia has it's own package manager which makes it easy to install packages and resolve dependencies.

The easiest way to install these is interactively, using the Julia REPL:

In the REPL, press the `]` key, which will bring up the package manager prompt[15]. Then type the install command:

[15] Press the `Left Arrow` key to return to the Julia prompt.

```
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.9.0-beta4 (2023-02-07)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

(@v1.9) pkg> add DifferentialEquations
```

This will install the package `DifferentialEquations` in the default environment (`@v1.9`)[16]. You can also create project environments to make your research fully reproducible by ensuring that the packages used, e.g., in a publication are the same as the ones used later, when you revisit the problem.

[16] At the time of writing that is version 1.9.

4 Basic Julia Syntax

5 Comparison to Matlab

6 Some Clever Tricks in Julia

6.1 Acknowledgment

Some of these tricks are taken from Julia Notes.

Part II

Examples

7 Solving ODEs and ODE systems

Solving Ordinary Differential Equations is one of the most common use cases for scientific computing in engineering applications.

The Julia package DifferentialEquations.jl is one of the biggest selling points of the language. It offers an unparalled range of solvers, all using the same interface[17].

[17] So changing the solver does not require any changes in the definition of the problem, even when moving between ODEs, DAEs and SDAEs.

7.1 Example: 4-Element Windkessel Model

The windkessel model is a common model for the pressure response of the vascular system (blood circulation) to a periodic, pulsing flow waveform (Westerhof et al. 2019).

Here we are going to work with the 4-Element windkessel model (Stergiopulos, Westerhof, and Westerhof 1999), comprising a flow source (time dependent flow rate), two resistors for characteristic Resistance of the near vessel (aorta), R_c, and systemic (peripheral) resistance, R_p, a compliance (capacitance) C, representing the blood storage capacity of the peripheral vessels, and an inductance L_p, representing the inertia in the proximal, large vessel, e.g., the aorta.

Westerhof, Nicolaas, Nikolaos Stergiopulos, Mark I. M. Noble, and Berend E. Westerhof. 2019. *Snapshots of Hemodynamics: An Aid for Clinical Research and Graduate Education*. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-91932-4.

Stergiopulos, Nikos, Berend E. Westerhof, and Nico Westerhof. 1999. "Total Arterial Inertance as the Fourth Element of the Windkessel Model." *American Journal of Physiology-Heart and Circulatory Physiology* 276 (1): H81–88. https://doi.org/10.1152/ajpheart.1999.276.1.H81.

The pressures in this circuit - p_1 before, and p_2 after the proximal L-R element - are described by the system of ODEs:

$$\frac{dp_1}{dt} = -\frac{R_c}{L_p}p_1 + \left(\frac{R_c}{L_p} - \frac{1}{R_p C}\right)p_2 + R_c\frac{dI(t)}{dt} + \frac{I(t)}{C} \quad (7.1)$$

$$\frac{dp_2}{dt} = -\frac{1}{R_p C}p_2 + \frac{I(t)}{C} \quad (7.2)$$

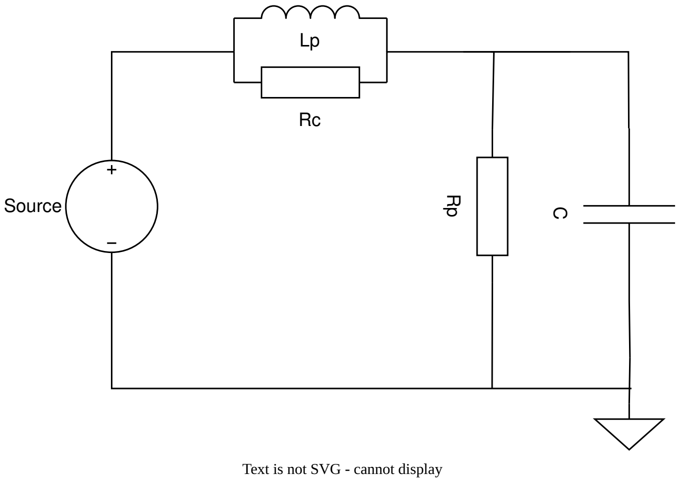

Figure 7.1: 4-Element Windkessel Model

In order to implement this model, we need to load the required modules. We use `DifferentialEquations`, `Plots`, and `ForwardDiff` for the time-derivative $\frac{\partial I}{\partial t}$:

```
using DifferentialEquations, ForwardDiff, Plots
```

The input waveform is a generic half-period of a sine-wave with a systolic (ejection) time of $t_{syst} = 0.4T$, with $T = 1$ s period-time (60 beats per minute). The dicrotic notch is modelled by running the sine into the negative for $t_{dicr} = 0.03$ s:

$$I = \begin{cases} I_{min} + (I_{max} - I_{min}) \sin\left(\frac{\pi}{t_{syst}} t\right) & \text{if } t < (t_{syst} + t_{dicr}) \\ I_{min} & \text{else} \end{cases} \tag{7.3}$$

In Julia, this function is implemented as[18]:

```
# max and min volume flow in ml/s
max_i::Float64 = 425
min_i::Float64 = 0.0

# period time
T::Float64 = 1.0

# Syst. Time in s
```

[18] Note that we use type specifications to define the parameters. Julia does suffer in performance, when untyped global variables are used, since these break type stability in the multiple dispatch. Making these parameter constant fixes their type. We should really be using these parameters in the function definition, or use a lambda function. But a lambda function is slower than typed variables.

```julia
systTime::Float64 = 2 / 5 * T

# Dicrotic notch time in s
dicrTime::Float64 = 0.03

function I(t)
    # implicit conditional using boolean multiplicator
    # sine waveform
    t = t - T * (t ÷ T)

    return ((max_i - min_i) * sin(pi / systTime * (t))
            * (t < (systTime + dicrTime) )
            + min_i)
end
```

We can quickly plot this function in Figure 7.2.

```julia
plotTime = LinRange(0,1,100)

plot(plotTime, I.(plotTime),
     xlabel = "t [s]", ylabel = "I [ml s<sup>-1</sup>]", label = "Outflow")
```

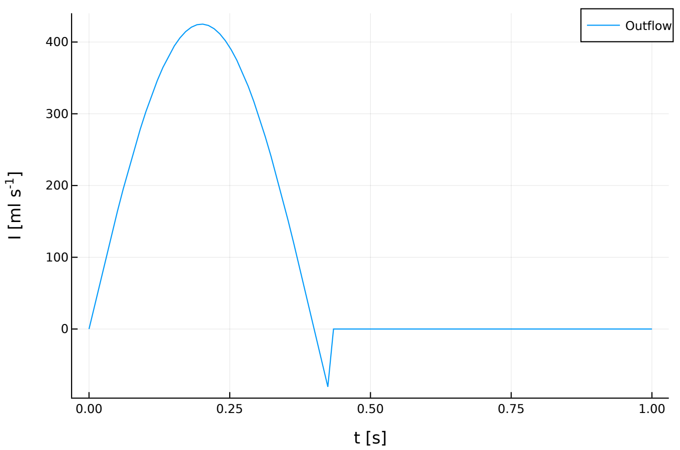

Figure 7.2: Generic waveform, representing ejection of blood from left ventricle in the aorta, including dicrotic notch (backflow at valve closure).

The definition of the ODEs Equation 7.1 and Equation 7.2 is done as a function with parameters `dP` and `P`, for $\frac{dp_{1,2}}{dt}$, and $p_{1,2}$, respectively[19]

[19] `P` is a vector of values, actually, as is `dP`.

```
function wk4(dP, P, params, t)

    Rc, Rp, C, Lp = params

    dP[1] = (
        -Rc / Lp * P[1]
        + (Rc / Lp - 1 / Rp / C) * P[2]
        + Rc * ForwardDiff.derivative(I, t)
        + I(t) / C
        )

    dP[2] = -1 / Rp / C * P[2] + I(t) / C

    return dP[1], dP[2]

end
```

We define the parameters, initial conditions, and time span for the integration:

```
Rc = 0.03
Rp = 1.0
C  = 2.0
Lp = 0.02

tspan = (0, 10)

params = [Rc, Rp, C, Lp]

P0 = zeros(2)
```

```
2-element Vector{Float64}:
 0.0
 0.0
```

And define the ODE problem and solve it[20]. We will time the run using the `@time` macro:

```
prob = ODEProblem(wk4, P0, tspan, params)

@time sol = solve(prob, DP5(), reltol=1e-9);
```

[20] We use the Dormand-Prince solver `DP5` here, because that is the same algorithm that Matlab's `ode45` uses. DifferentialEquations.jl has a multitude of other solvers that may perform better. Play around with these.

```
6.834239 seconds (10.99 M allocations: 795.646 MiB, 11.48% gc time, 99.98% compilation ti
```

Looking at this run time, we see that the run is slower than the Matlab run[21]. Looking at the details of the benchmark times, we see that most of that time has been used on compilation. So when we re-run the solver, it should take less time:

[21] See below. What happened here? Doesn't everybody say how much faster Julia is than Matlab?

```
@time sol = solve(prob, DP5(), reltol=1e-9);
```

```
0.001451 seconds (2.95 k allocations: 252.766 KiB)
```

And indeed, the run time is now one order of magnitude faster than the Matlab times shown in Section 7.1.1.

We can plot the solution in Figure 7.3 using the special plot recipe for ODE solutions:

```
plot(sol,
     label = ["p1" "p2"],
     xlabel = "t [s]",
     ylabel = "p [mm<sub>Hg</sub>]",
     tspan=(9,10))
```

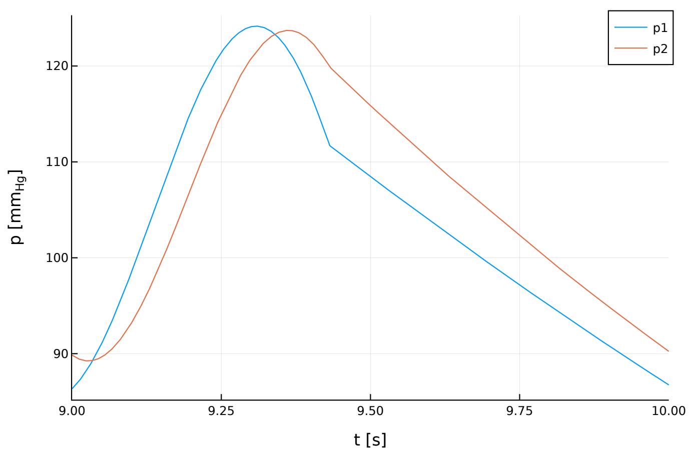

Figure 7.3: Solution of 4-element windkessel model using Julia's DifferentialEquations.jl

7.1.1 Comparison to Python and Matlab

For those coming from Python or Matlab, let's have a look at how this problem can be solved in these two languages and compare to the Julia version.

Switch between the languages using the tabs below:

7.1.2 Python

```python
import scipy as sp
from scipy import integrate
from scipy.misc import derivative

import numpy as np

import time

def wk4(t, y, I, Rc, Rp, C, Lp, dt):

    dp1dt = (
        -Rc / Lp * y[0]
        + (Rc / Lp - 1 / Rp / C) * y[1]
        + Rc * derivative(I, t, dx=dt)
        + I(t) / C
    )

    dp2dt = -1 / Rp / C * y[1] + I(t) / C

    return [dp1dt, dp2dt]

time_start = 0
time_end = 10

Rc = 0.2
Rp = 1.0
C = 1.0
Lp = 1e-2

dt = 1e-6

y0 = np.zeros(2)
```

```python
# Generic Input Waveform
# max volume flow in ml/s
max_i = 425

# min volume flow in m^3/s
min_i = 0.0

# Period time in s
T = 0.9

# Syst. Time in s
systTime = 2 / 5 * T

# Dicrotic notch time
dicrTime = 0.03

def I(t):
    # implicit conditional using boolean multiplicator
    # sine waveform
    I = (
        (max_i - min_i) * np.sin(np.pi / systTime * (t % T))
        *(t % T < (systTime + dicrTime)) + min_i
    )

    return I

tic = time.perf_counter()

sol = sp.integrate.solve_ivp(
    lambda t, y: wk4(t, y, I, Rc, Rp, C, Lp, dt),
    (time_start, time_end),
    y0,
    method="RK45",
    rtol=1e-9,
    vectorized=True,)

toc = time.perf_counter()

print(f"Elapsed time is {toc - tic:0.4f} seconds")
```

Runtime for this code is (timed using `time.perf.counter` in Python):

```
Elapsed time is 0.7359 seconds
```

7.1.3 MATLAB

```
tspan = [0, 10];

Rc = 0.03;
Rp = 1.0;
C = 2.0;
Lp = 1e-2;

P0 = [0, 0];

options        = odeset('Reltol',1e-9);

% Run once to allow Matlab to optimise
[t, P] = ode45(@(t,P) wk4(t,P,Rc,Rp,C,Lp), tspan, P0, options);

% Timed run
tic
[t, P] = ode45(@(t,P) wk4(t,P,Rc,Rp,C,Lp), tspan, P0, options);
toc



function dP = wk4(t,P,Rc,Rp,C,Lp)

dP     = zeros(2,1);

dP(1) = -Rc / Lp * P(1) ...
    + (Rc / Lp - 1 / Rp / C) * P(2) ...
    + Rc * didt(t) + i(t) / C;

dP(2) = -1 / Rp / C * P(2) + i(t) / C;

end

function i = i(t)

max_i = 425;
min_i = 0.0;
```

```
T = 0.9;

systTime = 2 / 5 * T;

dicrTime = 0.03;

i = ((max_i - min_i) * sin(pi / systTime * (mod(t,T))) ...
    *(mod(t,T) < (systTime + dicrTime)) ...
    + min_i);

end

function didt = didt(t)
dt = 1e-3;
didt = (i(t+dt) - i(t-dt)) / (2 * dt);
end
```

Runtime for this code is (timed using `tic toc` in Matlab)[22], which is a bit more than an order of magnitude slower than Julia:

```
Elapsed time is 0.030688 seconds.
```

So in this case, Julia is one order of magitude faster than Matlab and around 500x faster than Python[23] solving ODEs.

Personally, I find the Matlab code and, in particular, the Julia code easier to read.

[22] Same code in Octave, the free open-source version of Matlab runs in 3 seconds. Current versions of Matlab have improved runtime by partial just-in-time compilation. Note that the first run in Matlab is also slightly longer with 0.035 seconds, which is most likely due to Matlab optimising the solver to the problem.

[23] I have tried using PyPy and Cython in other cases and found that they speed up Python considerably. Unfortunately this was not the case when using SciPy and Numpy, which made the compiled Python version one order of magnitude **slower** than interpreted. There seems to be a problem with C-calls from PyPy.

References

A Julia Resources