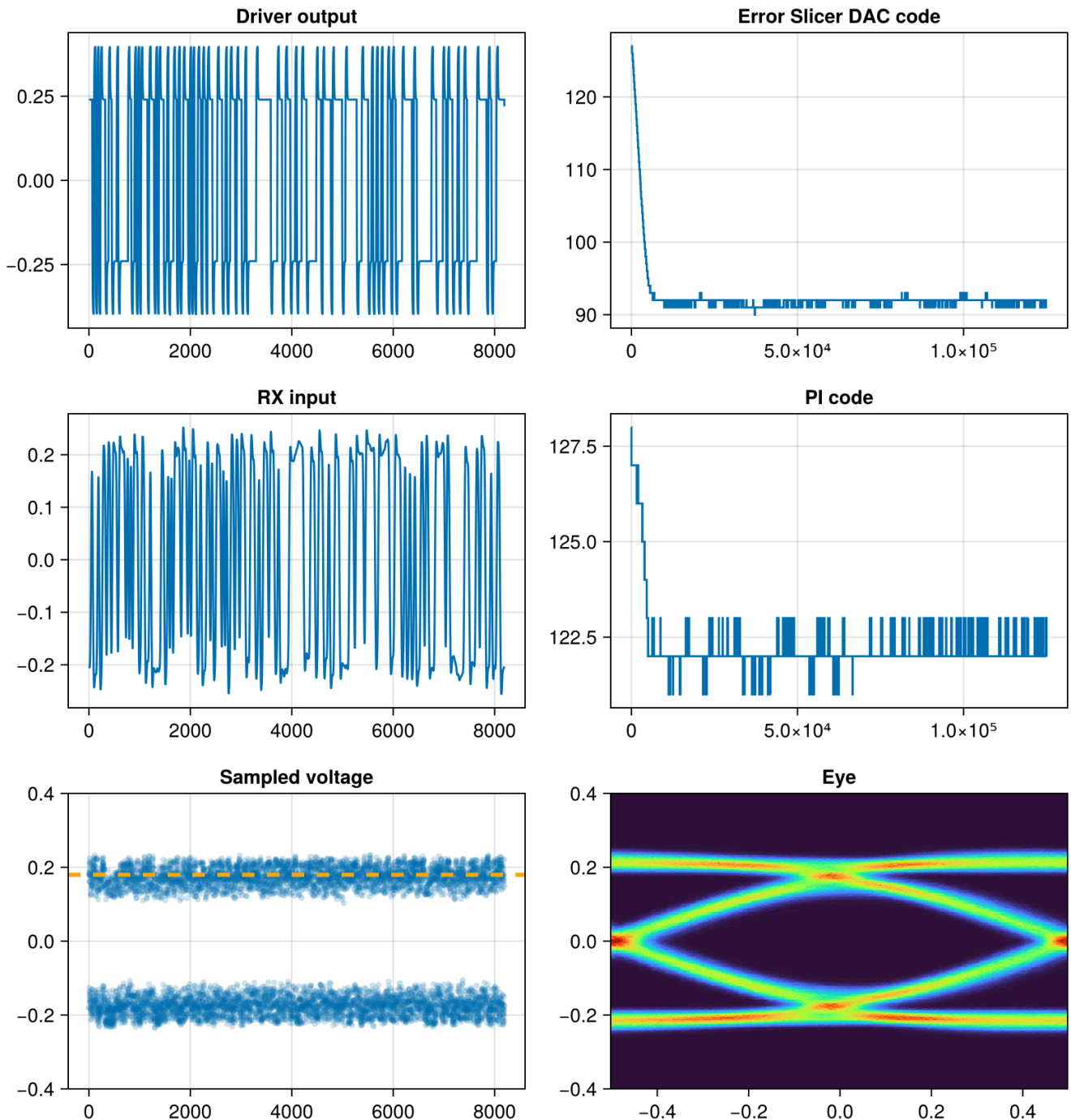


# Building SerDes Models in Julia, pt.1 - Background

---



This notebook shows how to build and simulate a "simple" SerDes model using Julia. It serves as a documentation of my own journey of learning Julia so far. Hopefully it can turn into something

bigger in the future (i.e., a more generic and expandable SerDes simulation framework). There will be several parts

1. Background and getting started on Julia
2. SerDes simulation framework
3. Data structures, math, and more
4. Detailed transmitter example
5. Plotting (w/ Makie)

Due to limited time and resource, the entire source code for this project won't be available yet on GitHub, but enough will be explicitly shown in the notebook to demonstrate the framework (which I believe is the most important part) and what Julia can do

## Why Julia?

---

While there are many languages already available for scientific computing, like Python and MATLAB, no one is without its fault. Below are some discussions on their pros/cons, and why I believe Julia is worth investigating

### Python

Despite its huge popularity, Python is almost a non-starter for me when it comes to building and simulating SerDes models. Python is a dynamic and interpreted language, which means the code is only converted to machine code at run time. Python's syntax is very human friendly and easy to read/learn, but this contributes to it being too slow compared to compiled languages like C/C++. Python is a great prototyping/scripting language, and when speed/performance doesn't matter as much, it's SWEET!

For computation heavy tasks, Numpy hides away Python's slowness by essentially pre-compiling all array data structures and operations in C code. However, speed of your model can then be bottlenecked by some new algorithm you are experimenting with, an explicit and necessary for-loop somewhere, or post-sim data analysis.

There are existing frameworks for SerDes modeling in Python, like SerDesPy and SymbaPy (not available on GitHub anymore). However, speed isn't the strongest point for these packages. SymbaPy quotes ~200 seconds for simulating 1 million bits (and ~2600 seconds w/ eye diagrams).

Nevertheless, Python's ease of entry and open-source community remains key in its wide adoption among academia and industry experts alike. Therefore, it's definitely not going away any time soon.



### MATLAB

I love MATLAB for its plotting and tooling features (open source packages are still far away from what MATLAB plots are capable of). Its revamped Just-in-Time (JIT) compiler since R2015 gives a good

trade-off between code performance and feeling "snappy" (like an interpreted language). MATLAB remains the go-to choice for industry and scientific communities due to its one-size-fit-all suite.

But well... it's not free (and I don't blame them for having so many great functions and features out of the box).

However, because of its closed source nature, the language made conscious decisions to eliminate concepts like "pointers" in C/C++, and manage variable passing internally. This could lead to memory inefficiency and allocation issues if the programmer is not careful. In the case of simulating millions of bits in a SerDes system, I often found MATLAB to be a big memory hog and simulation could begin to slow down as time goes on (due to garbage collection)

Simulink, SerDes Toolbox, IBIS-AMI compatibility, etc. are other big pluses that come with MATLAB. Yet, most simulation models are still custom built to evaluate proprietary architecture and algorithm. Besides, codes are often compiled again to speed up simulation. And finally, for someone with a soft spot for open-source, I find more joy in learning a new language than figuring out how to using new proprietary tools. 😊

## Julia

I came across Julia because I googled "open-source alternative to MATLAB" one day like I have done many times before.

### Julia in a Nutshell

#### Fast

Julia was designed for [high performance](#). Julia programs automatically compile to efficient native code via LLVM, and support [multiple platforms](#).

#### Dynamic

Julia is [dynamically typed](#), feels like a scripting language, and has good support for [interactive](#) use, but can also optionally be separately compiled.

#### Reproducible

[Reproducible environments](#) make it possible to recreate the same Julia environment every time, across platforms, with [pre-built binaries](#).

#### Composable

Julia uses [multiple dispatch](#) as a paradigm, making it easy to express many object-oriented and [functional](#) programming patterns. The talk on the [Unreasonable Effectiveness of Multiple Dispatch](#) explains why it works so well.

#### General

Julia provides [asynchronous I/O](#), [metaprogramming](#), [debugging](#), [logging](#), [profiling](#), a [package manager](#), and more. One can build entire [Applications and Microservices](#) in Julia.

#### Open source

Julia is an open source project with over 1,000 contributors. It is made available under the [MIT license](#). The [source code](#) is available on GitHub.

Essentially, Julia aims to solve the "two language problem" - we want a language that is easy/flexible like Python, but fast like C (especially for scientific computing). Julia's own introduction paragraph summarizes itself well:

### Julia Compared to Other Languages

Julia features optional typing, multiple dispatch, and good performance, achieved using type inference and just-in-time (JIT) compilation (and optional ahead-of-time compilation), implemented using LLVM. It is multi-paradigm, combining features of imperative, functional, and object-oriented programming. Julia provides ease and expressiveness for high-level

numerical computing, in the same way as languages such as R, MATLAB, and Python, but also supports general programming. To achieve this, Julia builds upon the lineage of mathematical programming languages, but also borrows much from popular dynamic languages, including Lisp, Perl, Python, Lua, and Ruby. [link here](#)

I then decided to migrate some of my MATLAB models to Julia and do a benchmark comparison. The porting is relatively straight-forward once I picked up Julia's typing system, and below are code snippets of a PRBS generator function in Julia and MATLAB

This is a Julia function

```
function bist_prbs_gen(;poly, inv, Nsym, seed)
    seq = Vector{Bool}(undef, Nsym)
    for n = 1:Nsym
        seq[n] = inv
        for p in poly
            seq[n] ⊔= seed[p]
        end
        seed .= [seq[n]; seed[1:end-1]]
    end
    return seq, seed
end
```

This is a MATLAB function

```
function [seq, seed] = bist_prbs_gen(poly, inv, Nsym, seed)
    seq = zeros(1, Nsym);
    for n = 1:Nsym
        seq(n) = inv;
        for p = poly
            seq(n) = xor(seq(n), seed(p));
        end
        seed = [seq(n), seed(1:end-1)];
    end
end
```

You can see the syntax between Julia and MATLAB is quite similar. The algorithm here is very straight-forward. Admittedly, I didn't do much optimization on the register and bit level as a first pass, but the goal here is to really test the capability of the compilers.

A couple of things to highlight

1. Julia allows special unicode characters as part of the code! The symbol  $\vee$  means "xor". Though it doesn't make a difference in the code functionality, it does make one happy when you can use  $\pi$  and  $e$  in your code.
2. There are some smaller nuances when dealing with Julia arrays. The `.=` syntax is not a typo, but rather a broadcasting operation similar to MATLAB's element-wise operations (more on this in the future).

3. Variable typing in Julia is optional. Like Python, Julia can dynamically type variables at runtime. However, for the more advanced users, defining variable types early in Julia is also possible and often helps with performance.

Now, let's talk about speed. If I benchmark the functions by generating and timing 10 million bits using PRBS<sub>31</sub> (i.e. `bist_prbs_gen(poly=[28,31], inv=false, Nsym=10e6, seed=ones(31))`), below are the results. I needed to make sure I didn't make a mistake, but yes there is an almost **10x** difference between these seemingly identical functions!

## JULIA

```
julia> @time bist_prbs_gen1(poly=[28,31],inv=false, Nsym=Int(10e6), seed=ones(Bool,31));  
1.792685 seconds (20.00 M allocations: 1.499 GiB, 13.81% gc time)
```

## MATLAB

```
>> tic; [~, ~] = bist_prbs_gen([28,31], false, 10e6, seed);toc  
Elapsed time is 15.648272 seconds.
```

For those interested, you can try implementing the same in Python. Surprisingly, the implementation using Python list instead of Numpy array is faster when I tried (~6s for list and ~12s for numpy arrays).

Assuming I didn't do anything blatantly false, this just adds to the confusion about how to optimize Python code when performance matters.

```

import time
import numpy as np

def bist_prbs_gen(poly, inv, nsym, seed):
    seq = [False]*nsym
    for n in range(nsym):
        seq[n] = inv
        for p in poly:
            seq[n] ^= seed[p-1]
        seed[1:] = seed[0:-1]
        seed[0] = seq[n]

    return seq, seed

def bist_prbs_gen_arr(poly, inv, nsym, seed):
    seq = np.array([False]*nsym)
    for n in range(nsym):
        seq[n] = inv
        for p in poly:
            seq[n] ^= seed[p-1]
        seed[1:] = seed[0:-1]
        seed[0] = seq[n]

    return seq, seed

if __name__ == '__main__':
    start = time.time()
    seq1, seed1 = bist_prbs_gen([28, 31], False, int(10e6), [True]*31)
    end = time.time()
    print(end - start)    ## Showed 5.537s

    start = time.time()
    seq2, seed2 = bist_prbs_gen_arr([28, 31], False, int(10e6), np.array([True]*3
1))
    end = time.time()
    print(end - start)    ## Showed 13.543s

```

To drive the speed point home for Julia, the plot at the very top was generated after processing **one million bits** using a model that includes a jittered transmitter, noisy channel, a simple RX with slicer adaptation and CDR loops. **The entire simulation only took ~10 seconds.**

Of course, this example might be a bit artificial, but the **order of magnitude** speed improvement alone was enough to push me to spend more time learning Julia, so here we are 😊.


# Getting Started with Julia

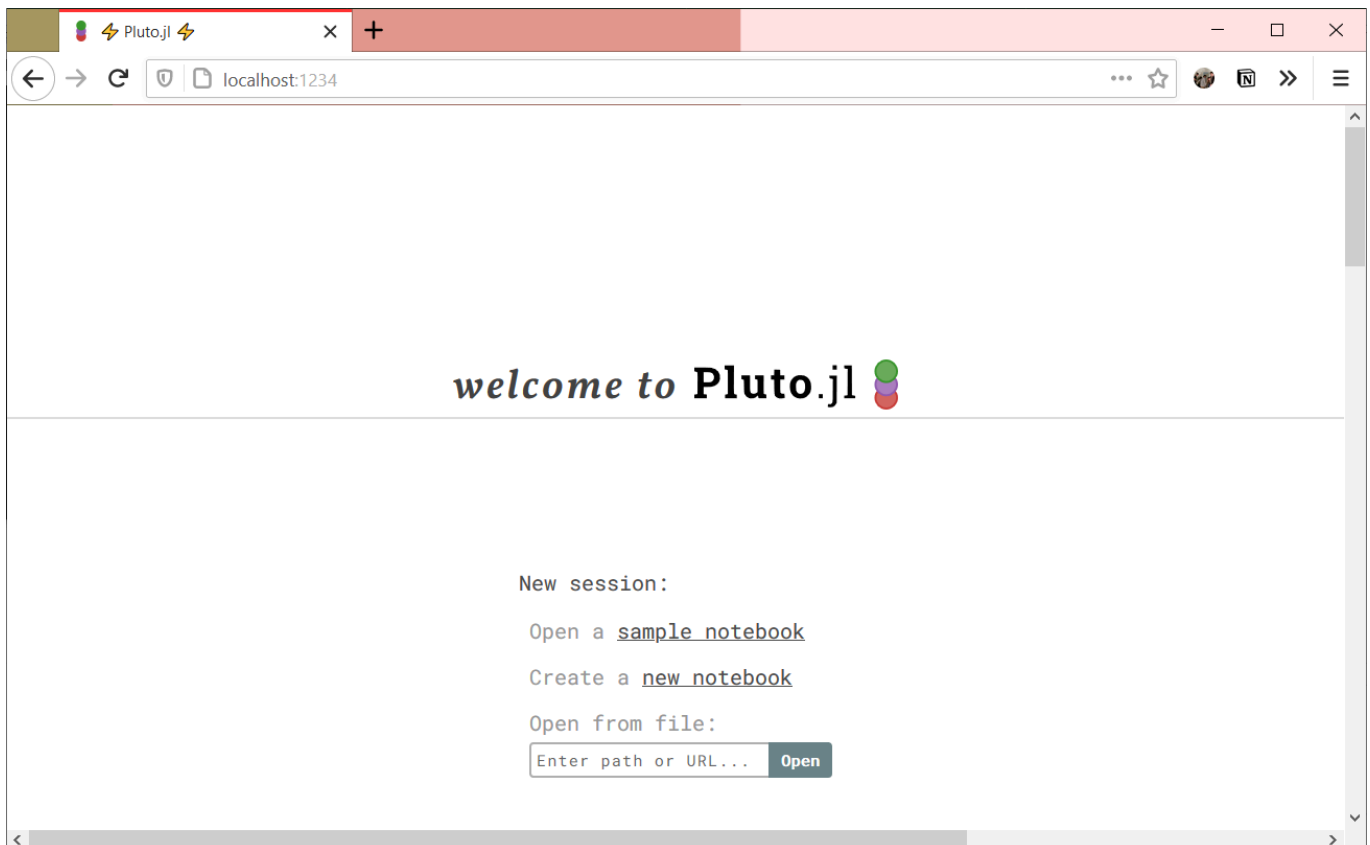
---

Julia is relatively young compared to Python, so development tools might look much more "basic". However, it still offers an easy installation and a good extension in [Visual Studio](#) (my current IDE choice).

If you want to go through my notebooks or you are intrigued by what Julia can do for your field, follow the steps [here](#) to install Julia and Pluto (VS Code is optional if you want to expand upon what are shown in the notebooks)

Once you can run Pluto from the Julia REPL like below, open this notebook using the GitHub URL (or copy the .jl file to your local director and open).

```
fons@woof: ~  
fons@woof:~$ julia  
 Documentation: https://docs.julialang.org  
Type "?" for help, "]"?" for Pkg help.  
Version 1.5.0-rc1.0 (2020-06-26)  
Official https://julialang.org/ release  
  
julia> import Pluto; Pluto.run()  
Go to http://localhost:1234/ to start writing ~ have fun!  
  
Press Ctrl+C in this terminal to stop Pluto
```



If everything goes well, you should be able to run the small *bist\_prbs\_gen* benchmark next.

The BenchmarkTools is the first helpful package we will learn to use. A macro is part of Julia's metaprogramming capability that simplifies code (think of it as a function manipulating the string of your code), denoted by the `@` sign. The `@time` macro is used to benchmark the execution time. What's neat is it also tells how many memory allocations are done, as well as compilation and garbage collection time.



Once you are used to it, it forces you to think a bit deeper down to the machine level, but you still remain writing in a high level language. **VERY SATISFYING!** 🥰

bist\_prbs\_gen (generic function with 1 method)

```
1 function bist_prbs_gen(;poly, inv, Nsym, seed)
2     seq = Vector{Bool}(undef,Nsym)
3     for n = 1:Nsym
4         for p in poly
5             seq[n] = seed[p]
6         end
7         seed .= [seq[n]; seed[1:end-1]]
8     end
9     return seq, inv, seed
10 end
```

```
1 using BenchmarkTools
```

```
1 @time bist_prbs_gen(poly=[28,31], inv=false, Nsym=Int(10e6), seed=ones(Bool,31));
```

```
1.306759 seconds (20.00 M allocations: 1.501 GiB, 6.55% gc time)
```



For those who are more familiar with Python, Julia's JIT compilation might feel weird at first because the run time will be longer for the first run (due to compilation time) and you get the real time for subsequent runs. It's something you need to get used to if you are accustomed to Python's consistent behavior after hitting run. Nevertheless, this in my opinion makes Julia more suitable for running long simulations with more complicated models rather than simple scripting on lab benches.

Once you are setup with Julia, I recommend introducing yourself to Julia through the following resources, and familiarize with these packages

- ["Why We Created Julia"](#)
- [Julia's official documentation](#)
- [BenchmarkTools.jl](#) - performance tracking
- [Revise.jl](#) - makes Julia feel more like a runtime language when code changes
- [UnPack.jl](#) - easier syntax to manipulate struct data
- [DataStructures.jl](#) - special data structures like buffers, queues, trees, etc.
- [Parameters.jl](#) - easier model parameter handling with default values and keywords
- [Interpolations.jl](#) - interpolation in Julia
- [DSP.jl](#) - fast DSP functions like convolutions
- [Makie.jl](#) - my choice of plotting package in Julia