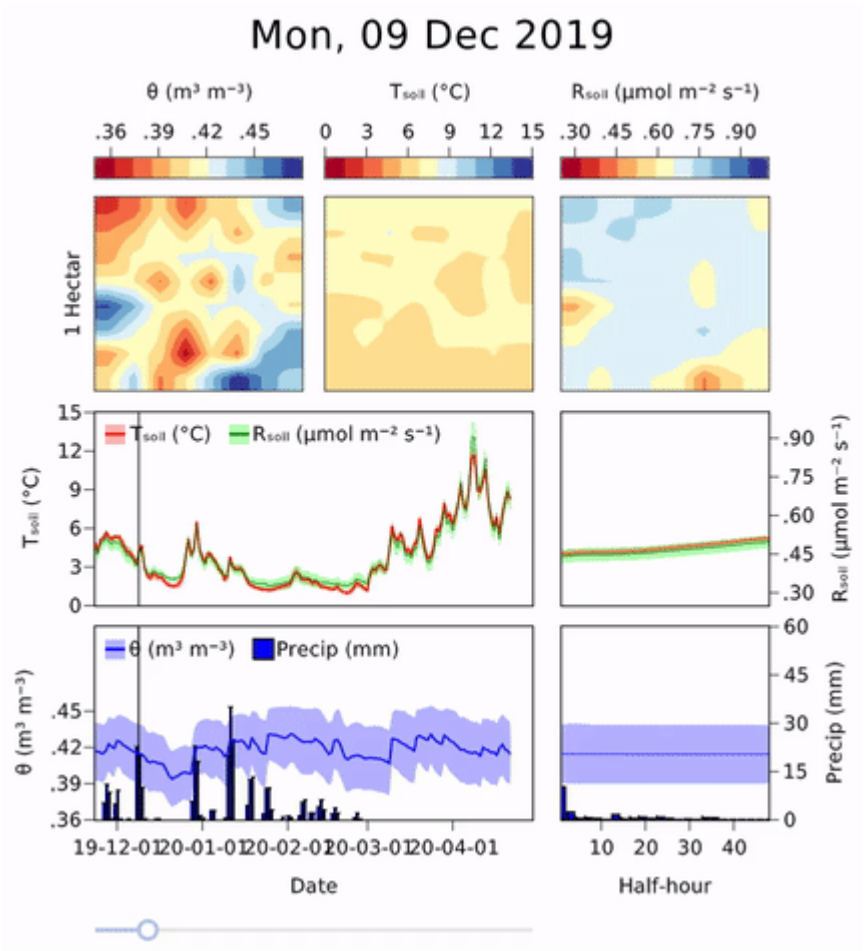
Building SerDes in Julia pt. 5A - Plotting and More



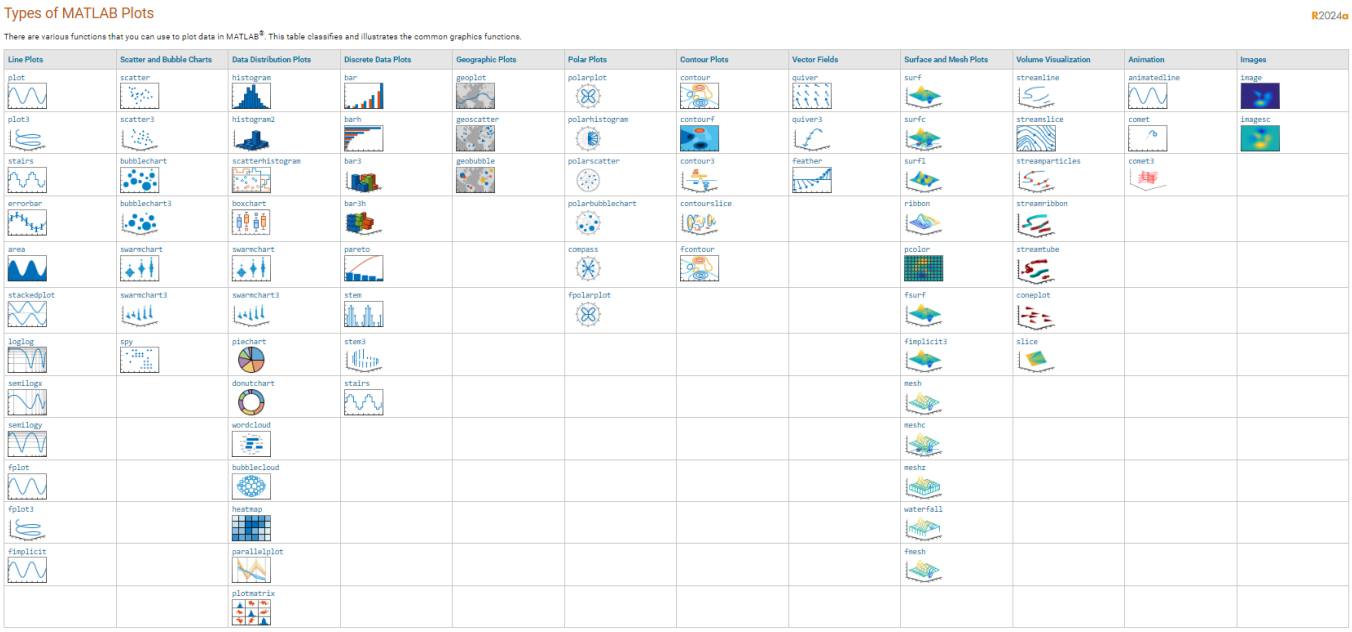
I can't remember when MATLAB plots first entered my life, but it was life changing to say the least. I still think MATLAB has the best plotting capabilities with great features and flexibility (that's why they are charging for it 💸), but it doesn't mean open-source plot packages have to suck.

In this notebook, I will briefly summarize what are the plotting options (yes, option with "s") in Julia, compare with MATLAB and Python's matplotlib, and why I think Makie might be the best option for our SerDes models (or the best overall in the long run). At time same time, we will introduce some of the most powerful features in Julia, including multiple dispatch, struct as functions, Observables, etc.

What makes MATLAB plots great

There are mainly three key things for any plotting package to be great

1. Publication-quality graphics (i.e., vector graphics, good color palletes, etc.)
2. Flexibility and ease of customization
3. Interactivity/user-friendly interface as a debug tool

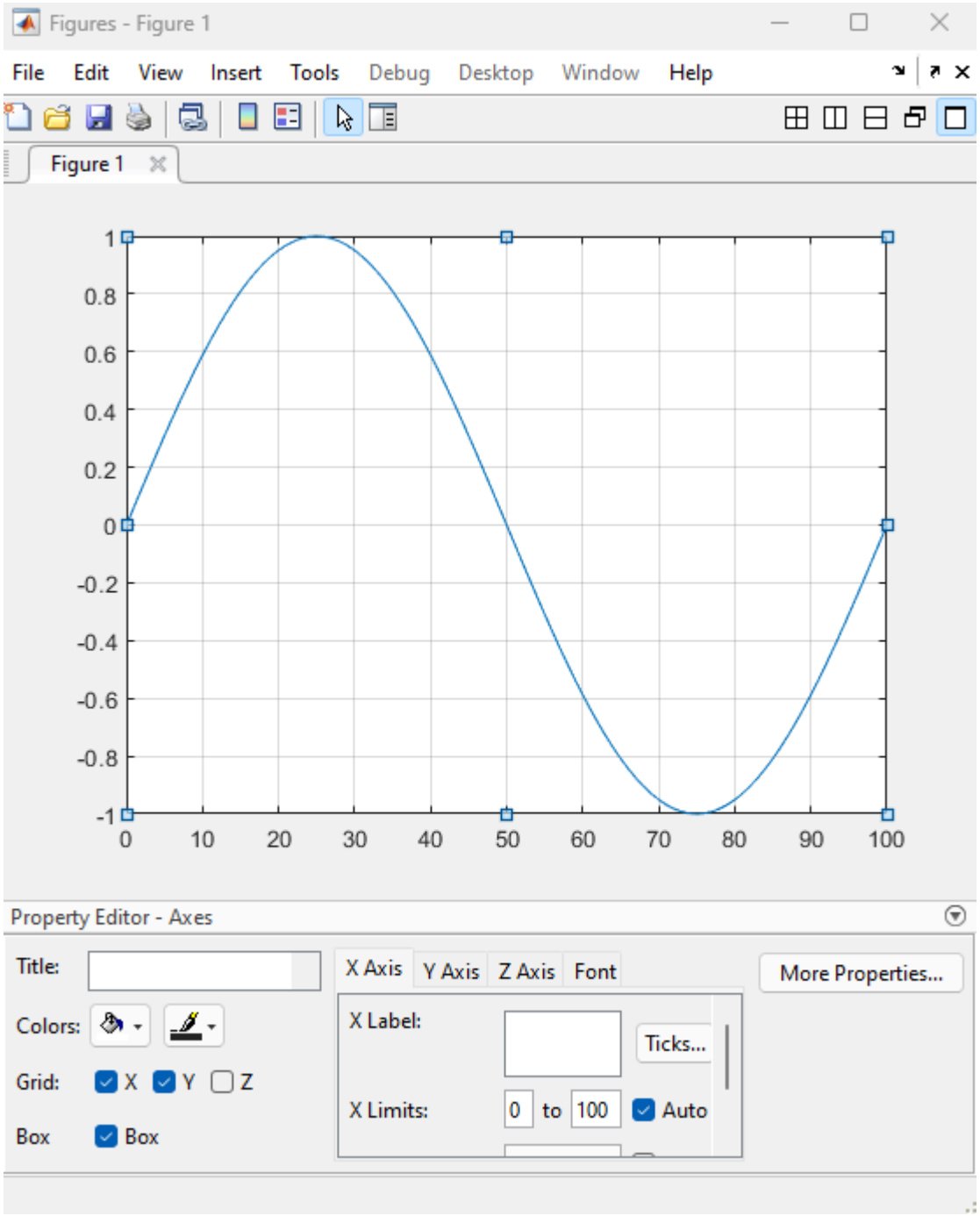


I would argue MATLAB has the best overall package considering these three points. MATLAB's capability of playing around with plots using just a **mouse** in a well designed Figure GUI remains the golden standard to me.

In particular, I often find myself spending more time using plots as debug tools. Let me know if you think the following MATLAB command history look familiar

```
Figure(1); grid on
plot(x,y)

Figure(2);
subplot(1,1,2); hold on;
plot(x2,y2, '-o')
plot(x3,y3, '-.')

subplot(1,2,2);
plot(a, b)

Figure(); plot(...)
plot(...)
plot(...)
...
```

MATLAB has made its "time to plot" very short with intuitive interface and command syntax. Just look at all these buttons and menus on the window! This meant that I didn't need to *code* how I want the plot to look like. After dropping in the data, I could just customize right in the figure window until it's ready for a PPT or paper.

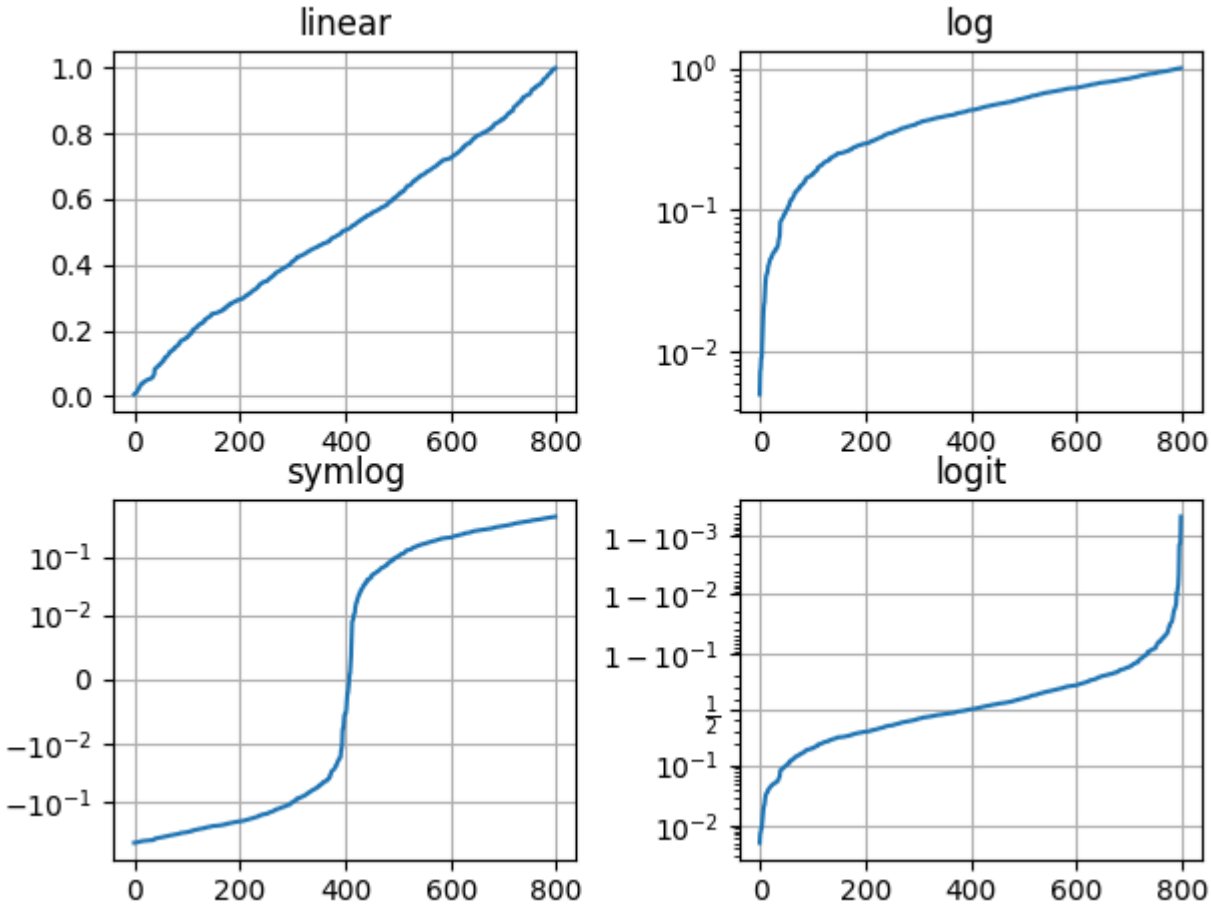


Matplotlib - the next best thing?

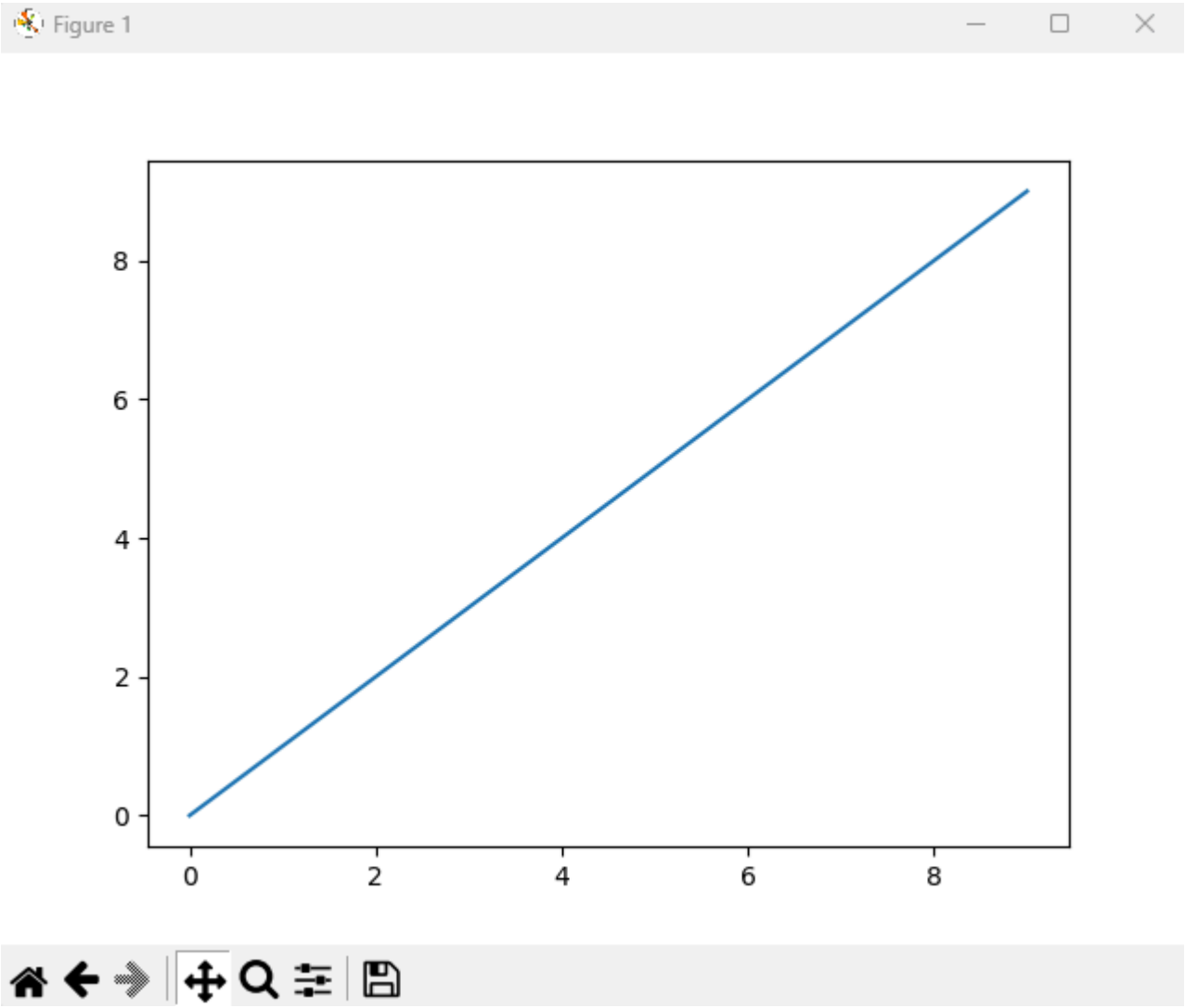
Because MATLAB's plot is so great (just to be fair and not anger any other plotting community, I heard great things about `ggplot` in R too, but I haven't personally used it), pretty much all other plotting packages in the open-source world mimic it.

Python's `matplotlib.pyplot` has evolved to be almost a MATLAB twin in terms of plot varierty and command syntax, as shown in this example from matplotlib's website

```
# Fixing random state for reproducibility
np.random.seed(19680801)

# make up some data in the open interval (0, 1)
y = np.random.normal(loc=0.5, scale=0.4, size=1000)
y = y[(y > 0) & (y < 1)]
y.sort()
x = np.arange(len(y))

# plot with various axes scales
plt.figure()

# linear
plt.subplot(221)
plt.plot(x, y)
plt.yscale('linear')
plt.title('linear')
plt.grid(True)

# log
plt.subplot(222)
plt.plot(x, y)
plt.yscale('log')
plt.title('log')
plt.grid(True)

# symmetric log
plt.subplot(223)
plt.plot(x, y - y.mean())
plt.yscale('symlog', linthresh=0.01)
plt.title('symlog')
plt.grid(True)

# logit
plt.subplot(224)
plt.plot(x, y)
plt.yscale('logit')
plt.title('logit')
plt.grid(True)
# Adjust the subplot layout, because the logit one may take more space
# than usual, due to y-tick labels like "1 - 10^{-3}"
plt.subplots_adjust(top=0.92, bottom=0.08, left=0.10, right=0.95, hspace=0.25,
                    wspace=0.35)

plt.show()
```



However, the interactivity of a pyplot is way worse than MATLAB figures. I still couldn't get used to how to zoom/pan/reset efficiently. While it has some keyboard shortcuts, I still find this window unintuitive to use (can't even double click to reset zoom?)



It's unfortunate that we might never get a MATLAB-like GUI when it comes to open-source plot packages. Perhaps we might continue to pay MATLAB's premium just for its plots (I have certainly been in situation where the data is processed or generated in another language and ported in MATLAB just for plots...).

Plotting in Julia faces similar challenges. Nevertheless, as a relatively new language, there could be more possibilities for a new winner to emerge. I will discuss the current state of Julia plots and highlight **Makie** in the next section.

State of Julia plots

As a language heavily advertised for data and scientific computing, Julia understood that it needs to have a good enough plotting ecosystem to attract people from MATLAB, R, Python, etc.

Instead of creating something new that still mimics MATLAB (like `pyplot`), Julia's default plot package, Plots.jl, adopts a unified plotting syntax BUT with options for different **backends**.

A backend determins the engine behind plot generations, and each backend may differ in its features and capabilities. The available backends and their advantages are well summarized in the official documentation.

At a glance

My favorites: `GR` for speed, `Plotly(JS)` for interactivity, `UnicodePlots` for REPL/SSH and `PythonPlot` otherwise.

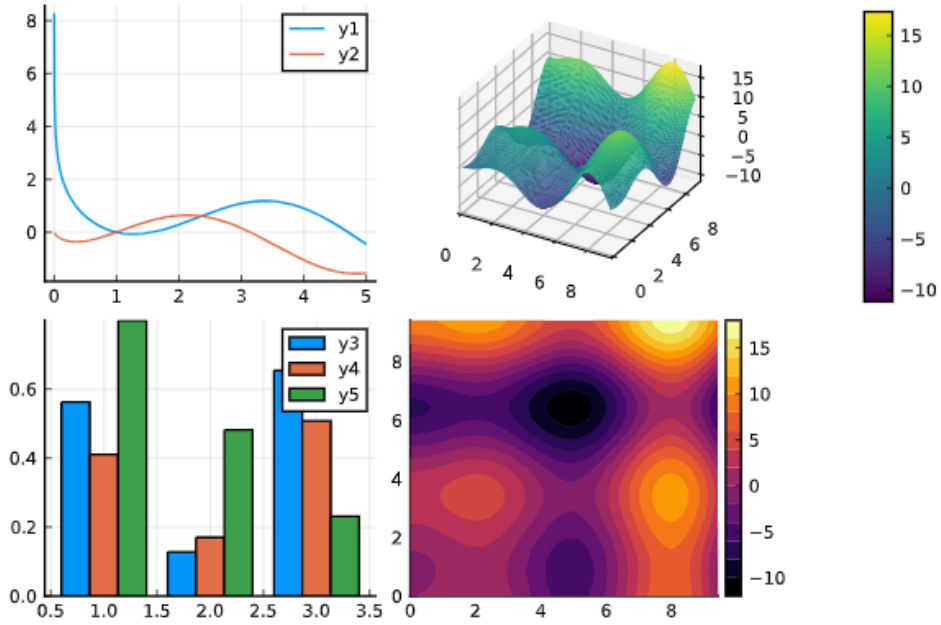
If you require...	then use...
features	GR, PythonPlot, Plotly(JS), Gaston
speed	GR, UnicodePlots, InspectDR, Gaston
interactivity	PythonPlot, Plotly(JS), InspectDR
beauty	GR, Plotly(JS), PGFPlots/ PGFPlotsX
REPL plotting	UnicodePlots
3D plots	GR, PythonPlot, Plotly(JS), Gaston
a GUI window	GR, PythonPlot, PlotlyJS, Gaston, InspectDR
a small footprint	UnicodePlots, Plotly
backend stability	PythonPlot, Gaston
plot+data -> `.hdf5` file	HDF5

Of course this list is rather subjective and nothing in life is that simple. Likely there are subtle tradeoffs between backends, long hidden bugs, and more excitement. Don't be shy to try out something new !

The eagle-eyed among you might have already spotted that PythonPlot is a supported backend. This means MATLAB/Python users could directly pick up from here when migrating to Julia (with small syntax tweaks). However, the GUI window issue persists (the same `pyplot` window will show up)

PythonPlot

A Julia wrapper around the popular python package `Matplotlib`. It uses `PythonCall.jl` to pass data with minimal overhead.



Pros:

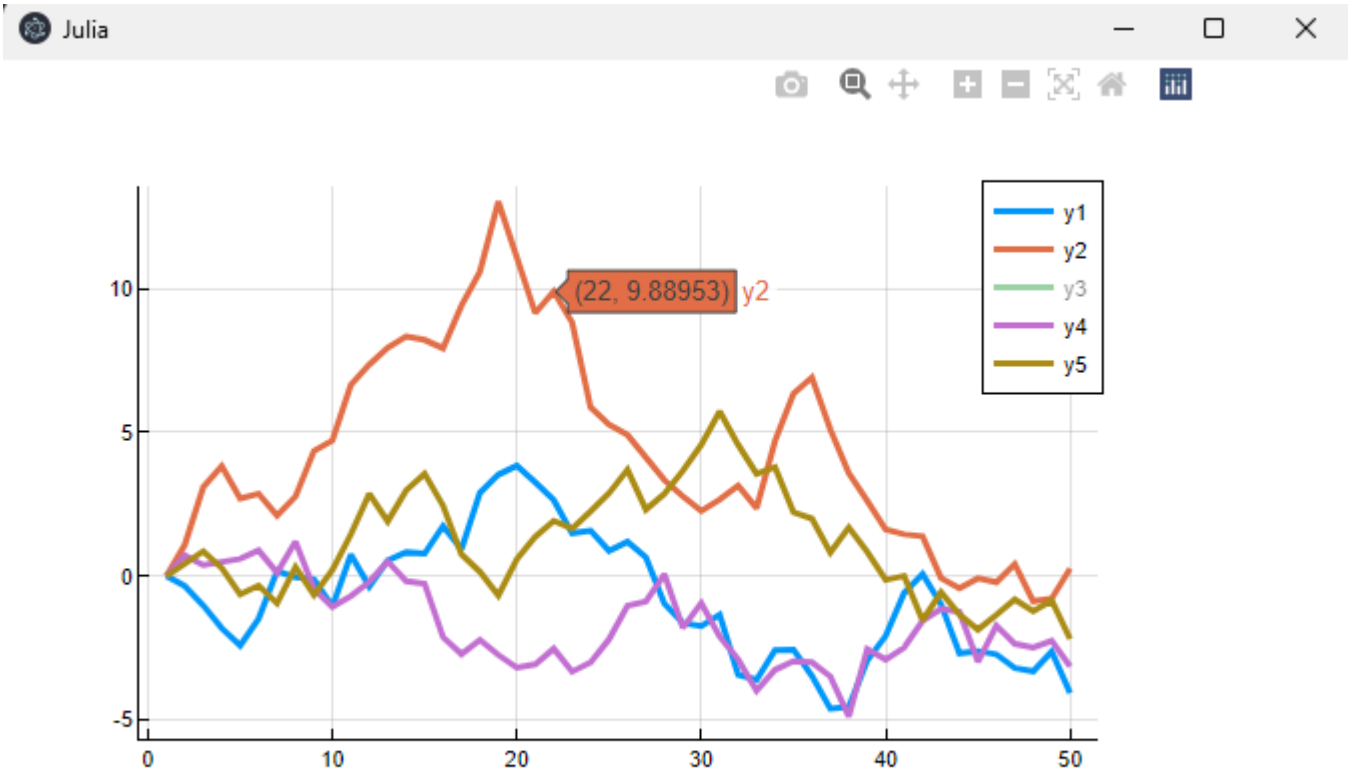
- Tons of functionality
- 2D and 3D
- Mature library
- Standalone or inline
- Well supported in Plots

Cons:

- Uses Python
- Dependencies frequently cause setup issues

Primary author: Steven G Johnson (@stevengj)

To address the interactivity issue in `pyplot`, PlotlyJS is a great backend for the best zooming/panning exeperience (you know it's more user-friendly when it's powered by Javascript). It also comes with axis resizing and legend toggling too (note the y3 trace is grayed out).



We don't have enough space and time to go over all the options in JuliaPlots here (again, Julia's goal initially is trying to be compatible and provide familiar experiences to as many new users as possible). In the case of plotting, I believe too many options is a good thing in an open-source environment.

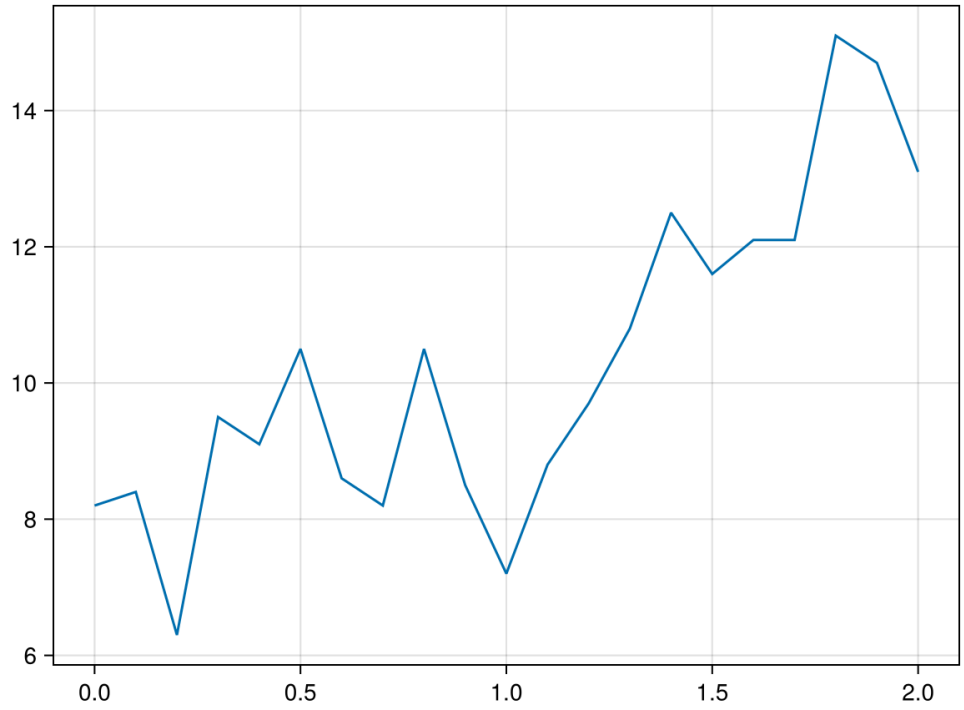
With this said, outside of the "official" Plots ecosystem, Julia has another dark horse (and definitely on the rise) plotting package called Makie. I will now detail my experience with Makie so far, and why this is my choice for the SerDes model visualization.

Makie - my choice for plotting in Julia

Compared to MATLAB and many of the open-source backends, Makie just feels different.

Syntax-wise, it is a more object based framework in which we can have control on **ALL** aspects of a plot element, including axis, legend, and the plot object itself. MATLAB and `matplotlib` also support this way of plotting, but normally we are more familiar with the implicit (or function based) framework (commands, commands, commands).

Nevertheless, Makie's plots are the best looking out of the box in my opinion (color choice, line thickness, axis fonts, grid, etc.). This saves me a lot of time from configuring the defaults like I have to do in MATLAB for instance.

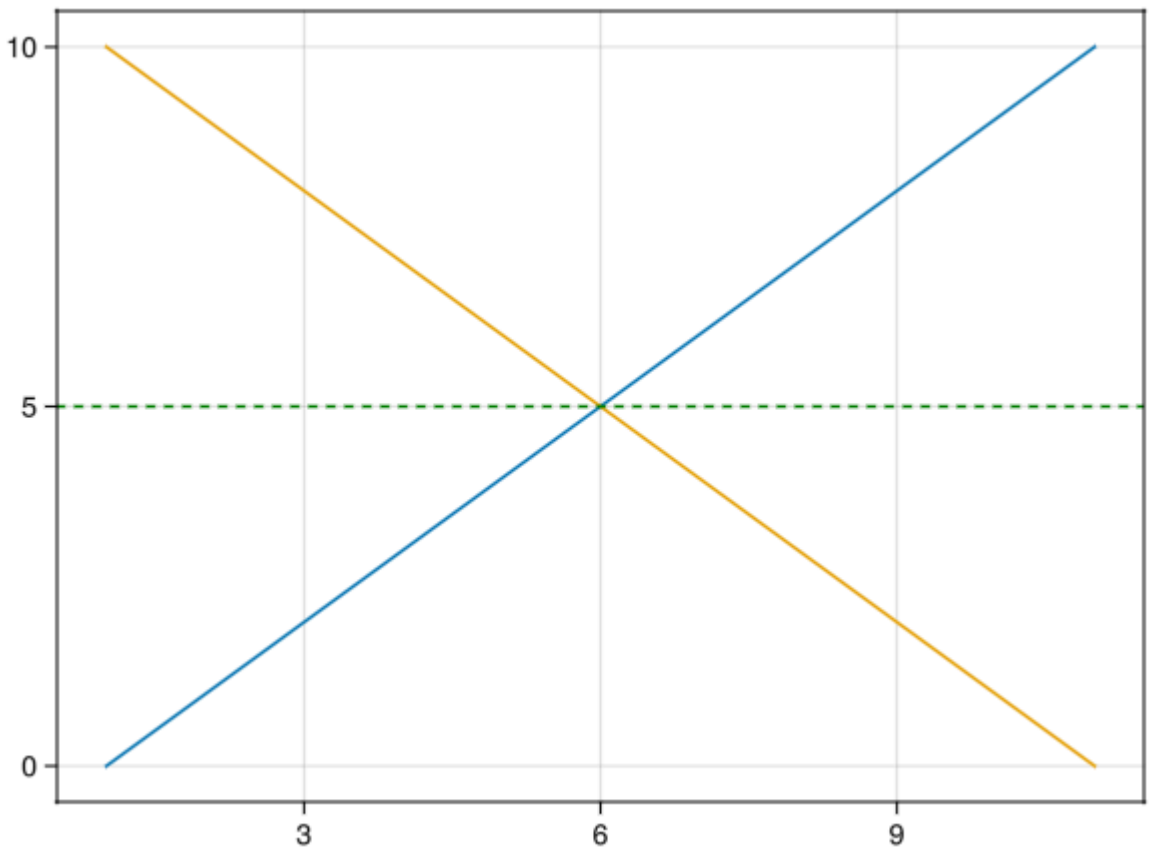


Similar to JuliaPlots, Makie also provide different backends. The most widely used are `GLMakie` (for interactive GUI window) and `CairoMakie` (for static publication-quality vector graphics). Another interesting and unique backend it provides is `WGLMakie`, basically `GLMakie` running in a browser! This becomes interesting when one wants to create customer-facing dashboards and demos, but `WGLMakie` isn't as mature as the other two. For our purpose, I will focus on GLMakie because interactivity is too important for me.

The "bad" about GLMakie (and how I "solve" it)

I will start with the "bad" about Makie/GLMakie: **it's too verbose**.

Makie thrives in flexibility - it has almost full extensibility and it's well-written enough that the behavior is quite predictable. However, the syntax is not (yet) for MATLAB people. Take the examples below



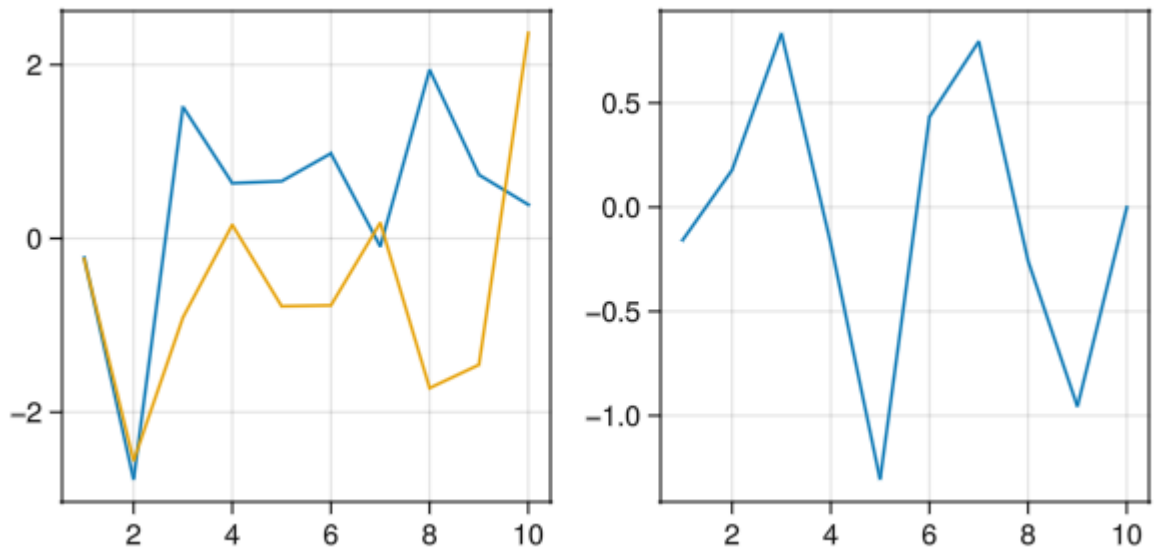

```
begin
f1, ax1, l1 = lines(0:10) #creates a new figure, along with the axis and line object
lines!(ax1, 10:-1:0)
hlines!(5, color=:green, linestyle=:dash) #no explicit axis given
f1
end
```

Each plotting function (in this case `line`) returns a `Figure (f1)`, `Axis (ax1)` and `Line (l1)` object (or other plot types like `Scatter` and `Stem`). It's quite important to "keep track" of these objects because that's what allows us to do full modifications.

A very Julian thing is to do is add ! to functions, so naturally every plotting function has a ! variant. The mutating `lines!` function takes an axis object, and draws onto the same axis without overwritting existing plots.

But wait, if I don't provide an explicit axis object, it still plots? That's because Makie plots into the `current_axis()`, and normally that's the last plot. So what's the point of `lines`? I now have to remember when to use `lines` and `lines!`?

To make matter worse, `lines` can also take an axis-like object, called `GridPosition`, and it's accessed by indexing the `Figure` object like. Ok, so `lines` plots into a `GridPosition`, but `lines!` plots into `Axis`? When I first tried Makie, I was very confused.




```
begin
f2 = Figure(size=(600,300));
ax2, l2 = lines(f2[1,1],randn(10))
lines!(Axis(f2[1,2]), randn(10))
lines!(ax2, randn(10))
f2
end
```

This "complicated" object framework alone was enough to almost push me away, but I am glad I persisted. Let's highlight some potential good here. The subplot management is built into Makie itself. Instead of explicitly specifying `subplot(x,y,n)` like in MATLAB, here we just begin plotting right away to the right place. The ! is equivalent to `hold on` in MATLAB, which helps ensure we don't accidentally lose a previous plot. There are also other goodness with Makie, but I just needed to find a way to plot more quickly like in MATLAB. So as a new Julian, I decided to solve this the Julian way - creating callable structs and using multiple dispatch.

Figz

Allow me to introduce `Figz` - a small wrapper around the most used Makie objects for easier plotting syntax (this is not complete yet since I believe it could be expanded further, but you could just copy this mini version here to get started).

The desired behaviors are the following

1. Each time a new `Figz` is constructed, a new window shows up, like MATLAB's Figure()
2. Seemlessly plots into subplots, but without explicitly creating `Axis`
3. Only mutating plot functions are necessary (although the full Makie extensibility should still be available)

```
begin
mutable struct Figz
    fig::Makie.Figure
    nrow::Int64
    ncol::Int64
    axes::Matrix{Makie.Axis}

    Figz(nr=1, nc=1; name="", datainspector::Bool=false, kwargs...) =
    (
        GLMakie.activate!(title=name);
        f = Figure(; kwargs...);
        display(GLMakie.Screen(),f);
        if datainspector
            DataInspector(f);
        end;
        axes = Matrix{Axis}(undef,nr,nc);
        for r=1:nr, c = 1:nc
            axes[r,c] = Axis(f[r,c])
        end;
        new(f, nr, nc, axes);
    )
end

function (figz::Figz)()
    current_figure!(figz.fig)
    current_axis!(figz.axes[1,1])
    return figz.fig
end

function (figz::Figz)(r=1,c=1)
    nrow = figz.nrow
    ncol = figz.ncol
    #if outside of current grid
    if r> nrow || c> ncol
        nrow_new = max(nrow,r)
        ncol_new = max(ncol,c)
        axes_new = Matrix{Makie.Axis}(undef, nrow_new, ncol_new)
        #copy to new axes matrix
        for n = 1:nrow, m=1:ncol
            if isassigned(figz.axes, n, m)
                axes_new[n,m] = figz.axes[n,m]
            end
        end
        #add new axes
        axes_new[r,c] = Axis(figz.fig[r,c])
        figz.axes = axes_new
        figz.nrow = nrow_new
        figz.ncol = ncol_new
    #if in current grid but no axis yet
    elseif ~isassigned(figz.axes, r, c)
        figz.axes[r,c] = Axis(figz.fig[r,c])
    end

    current_figure!(figz.fig)
```

```
    current_axis!(figz.axes[r,c])
    return figz.axes[r,c]
end


Base.getindex(x::Figz, args...) = (
    return getindex(x.fig, args...)
)

Makie.display(figz::Figz) = display(figz.fig)

end
```

Let's explain the code block by block. `Figz` is just a wrapper struct that contains the Makie figure's `Figure` and `Axis` objects. As of now, `Figz` assumes the simplest use case of subplots, each containing a single axis. The constructor takes optional number of row (`nr`) and number of column (`nc`) arguments to pre-define the subplot grid, but it's not that important since Makie does a great job figuring out the layout as we go. The keyword args for Makie's figure creation can also be directly passed through `kwargs`.

Each figure window can take an optional name and enable data inspector (GLMakie will show data points as mouse move over curves). Internally, all axes are generated and stored in a Matrix. We will then simply access these axes through a simpler interface.

```
mutable struct Figz
    fig::Makie.Figure
    nrow::Int64
    ncol::Int64
    axes::Matrix{Makie.Axis}

    Figz(nr=1, nc=1; name="", datainspector::Bool=false, kwargs...) =
    (
        GLMakie.activate!(title=name);
        f = Figure(; kwargs...); #creates the figure object
        display(GLMakie.Screen(),f); #creates a new window
        if datainspector
            DataInspector(f); #enable datainspector
        end;
        axes = Matrix{Axis}(undef,nr,nc);
        for r=1:nr, c = 1:nc
            axes[r,c] = Axis(f[r,c])
        end;
        new(f, nr, nc, axes); #construct a new Figz. Argument order matters
    )
end
```

Now here is the fun part: a Julia struct can behave as a callable function! We have seen this before: packages like `Interpolations` utilize this feature to create reusable interpolator object. Our first function will bring the particular figure (and by default the [1,1] axis) into focus, and return the `fig` object for further usage if needed. This allows easier subsequent plot syntax without explicitly specifying the figure/axis.

```
function (figz::Figz)()
    current_figure!(figz.fig)
    current_axis!(figz.axes[1,1])
    return figz.fig
end
```

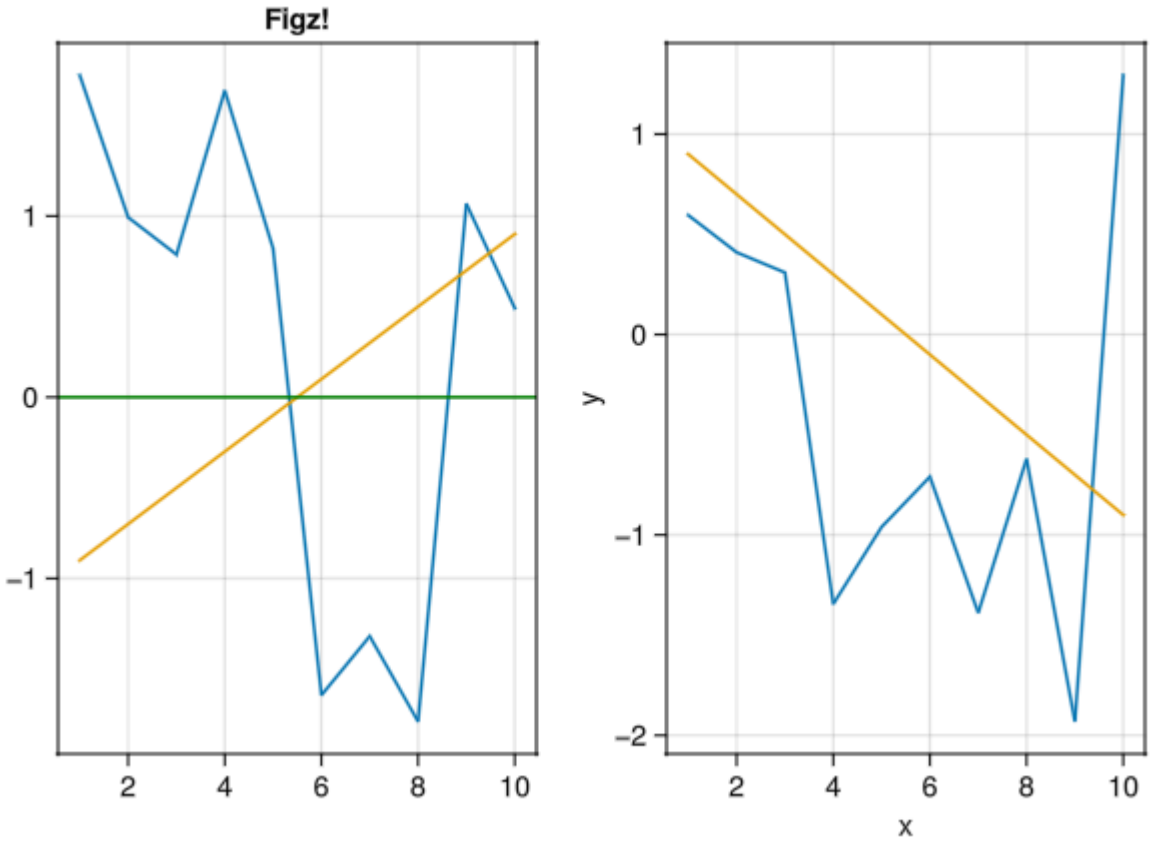
Julia tips

Julia structs can behave like functions with `function (x::T)(args...)` where `T` is the struct type. Multiple variations can be created simply with different arguments through the power of multiple dispatch.

Our next function will take two number, corresponding to row and col. This one will check if this axis exists in the current figure. If not, we use Makie to create and we store into the updated axes matrix. We then focus onto this figure and axis for plotting, and return the desired axis object for further usage.

```
function (figz::Figz)(r=1,c=1)
    nrow = figz.nrow
    ncol = figz.ncol
    #if outside of current grid
    if r> nrow || c> ncol
        nrow_new = max(nrow,r)
        ncol_new = max(ncol,c)
        axes_new = Matrix{Makie.Axis}(undef, nrow_new, ncol_new)
        #copy to new axes matrix
        for n = 1:nrow, m=1:ncol
            if isassigned(figz.axes, n, m)
                axes_new[n,m] = figz.axes[n,m]
            end
        end
        #add new axes
        axes_new[r,c] = Axis(figz.fig[r,c])
        figz.axes = axes_new
        figz.nrow = nrow_new
        figz.ncol = ncol_new
    #if in current grid but no axis yet
    elseif ~isassigned(figz.axes, r, c)
        figz.axes[r,c] = Axis(figz.fig[r,c])
    end

    current_figure!(figz.fig)
    current_axis!(figz.axes[r,c])
    return figz.axes[r,c]
end
```

With these two functions, we can already simplify Makie's syntax (for the quick plot case at least) like below. I believe this now is a more familiar experience to MATLAB, and to some extent feels simpler too.



Makie Tips

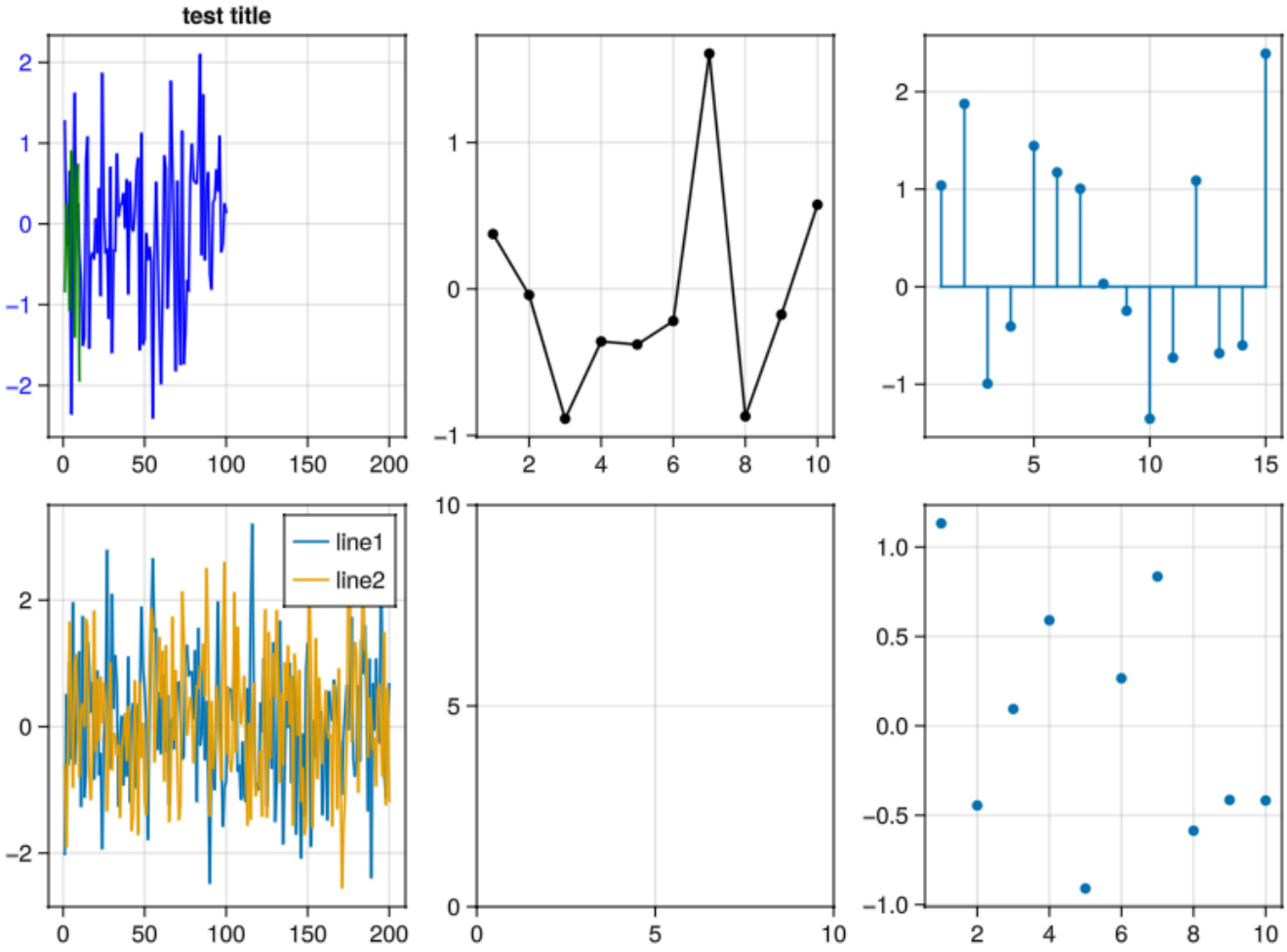
GLMakie's interactivity control is very intuitive. Use mouse left button to zoom and right button to pan. Ctrl + left-click will reset the zoom to full. Being an open-source ecosystem, there are new features requested everyday that takes the best of all tools out there. A recent pull request is related to toggling legend entry to hide/show curves like PlotlyJS. The upside of Makie is limitless.

Another thing I want to highlight that's possible in Julia, but not so easily (or just impossible) in other language is **multiple dispatch**. Without going into too many details, multiple dispatch allows us to override and define existing functions for our own data types. It's different from operator overloading (like in C++ or Python) where the function has to be called from an object. In Julia, the right function with the right argument types will be called automatically.

In this example, the `Figz` type has two such functions. `getindex` is the base function called when `[]` are used to access index. Here we basically allow a new syntax of `figz[x,y]` and it behaves the same as `fig[x,y]`. Similarly, we can override Makie's `display` to directly take our `Figz` type as well.

```
Base.getindex(x::Figz, args...) = (
    return getindex(x.fig, args...)
)

Makie.display(figz::Figz) = display(figz.fig)
```

So in conclusion, Makie's biggest problem for me is resolved with some small custom code. Use the playaround below to experiment with the `Figz` struct and its interface.




```
#play around with Figz here! New windows will pop up everytime you run this code cell
begin
fkz = Figz(1,2, datainspector=true, backgroundcolor = :white, size = (800, 600));

fkz(1,1).title = "test title"
fkz(1,1).yticklabelcolor = :blue
lines!(fkz(1,1), randn(100), color=:blue);


lines!(fkz(2,1), randn(200), label="line1");
lines!(randn(200), label="line2"); #once [2,1] is focused, we can keep plotting
axislegend() #legend will also be created for the focused axis

linkxaxes!(fkz(2,1), fkz(1,1));

scatter!(fkz(2,3), randn(10))
stem!(fkz(1,3), randn(15));


fkz2 = Figz()
lines!(randn(10))

fkz()
lines!(randn(10), color =:green)

```

```
26 fkz(1,2)
27 scatterlines!(randn(10), color =:black)
28
29 ax_new = Axis(fkz[2,2]) #I can still create a new axis the Makien way. Figz behaves
   the same as a Makie Figure because we overrode the getindex function. Note that this
   creation doesn't update Figz object's internal axes matrix. Only do this if needs
   twin axis (Maybe will be a built-in capability of Figz in the future)
30
31 fkz.fig #comment this two lines to see pop-up windows
```

The GOOD about Makie - `Observables`

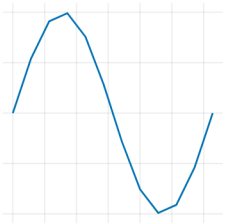
The one feature that ultimately made me stick with Makie is its compatibility with `Observables`. As a preview, please first check out the demo in this video.

It's amazing how Makie can create such cool animations with not that much code. Why does this interest me? In the time-domain SerDes model simulation, it's nice to have a figure that periodically updates waveform, parameters or eye diagrams that we are interested in. We can even record this as a video while the simulation runs, grab a coffee and we won't have missed a thing!

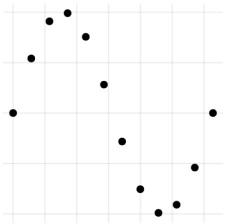
We don't have enough time here to do justice to all the other great features in Makie. Cheat sheets are provided below for easy reference. We will continue the topic of Observables in the next notebook, through an example of eye diagram animation.

Plotting Functions in Makie.jl :: **CHEAT SHEET**

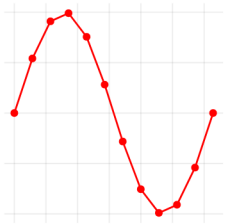
lines(x, y)



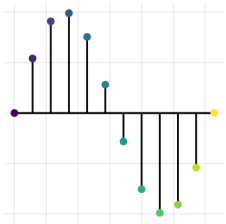
scatter(x, y)



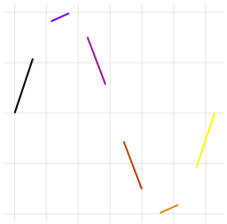
scatterlines(x, y)



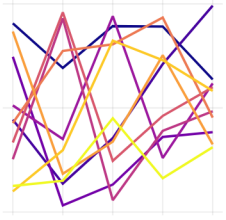
stem(x, y)



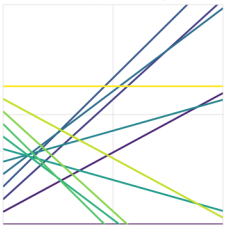
linesegments(points)



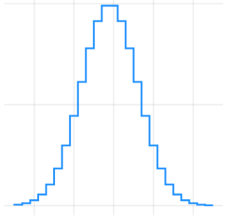
series(curves)



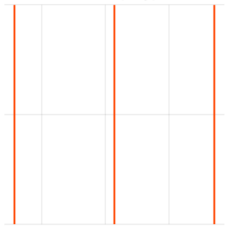
ablines(inter, slopes)



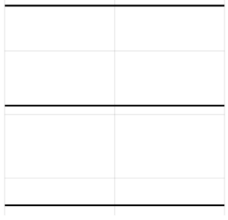
stairs(x, y)



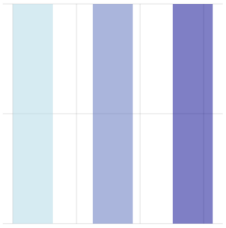
vlines(x, y)



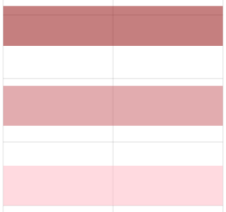
hlines(x, y)



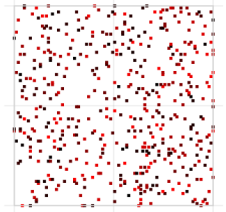
vspan(xlow, xhigh)



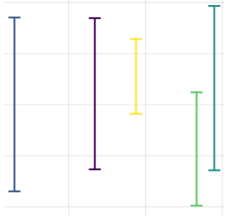
hspan(ylow, yhigh)



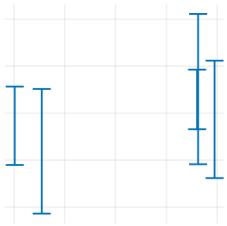
spy(x, y, sparseArray)



rangebars(x, low, high)

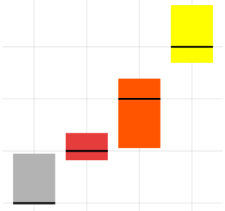


errorbars(x, y, yerr)

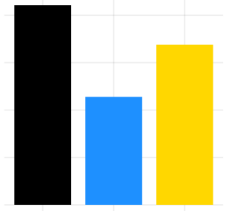


band(x, y-σ, y+σ)

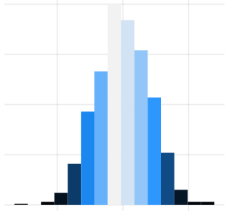
crossbar(x,y,ymin,ymax)



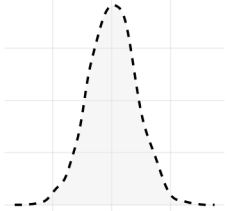
barplot(x,y)



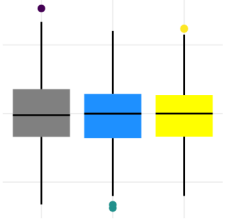
hist(x)



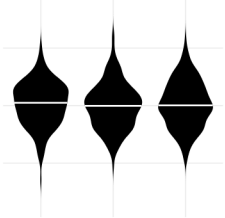
density(x)



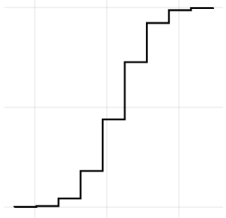
boxplot(x, y)



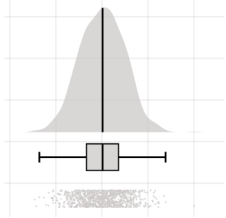
violin(x, y)



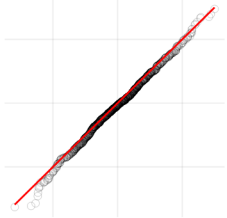
ecdfplot(x)



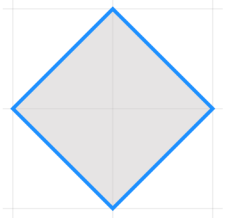
raincloudsplot(cat, y)



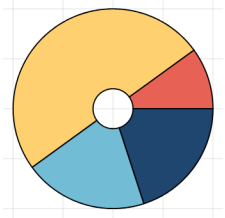
qqplot(x, y)



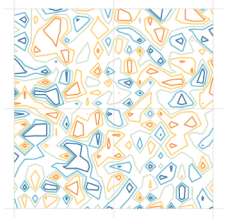
poly(points)



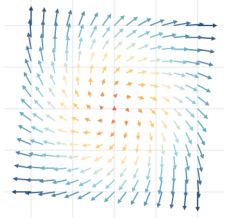
pie(fractions)



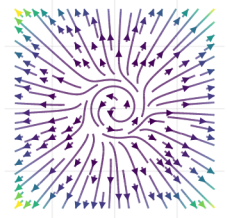
contour(x,y,vals)



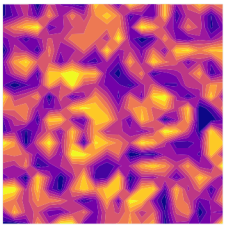
arrows(x,y,u,v)



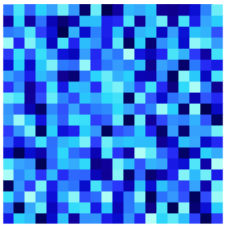
streamplot(f, x, y)



contourf(x,y,vals)



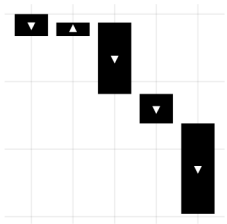
heatmap(x,y,vals)



image(x,y,img)

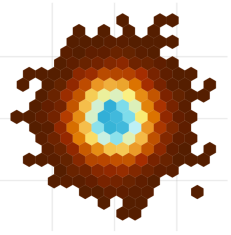
mesh(vertices,faces)

waterfall(x, y)

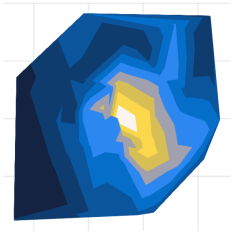


Plotting Functions in Makie.jl :: **CHEAT SHEET**

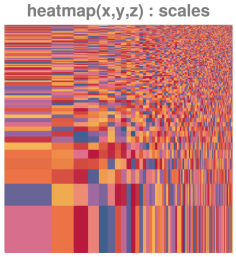
hexbin(x, y)



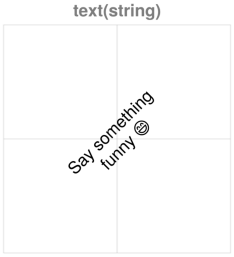
tricontourf(x,y,z)



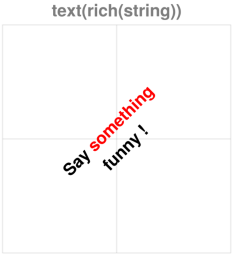
heatmap(x,y,z) : scales



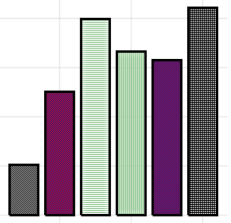
text(string)



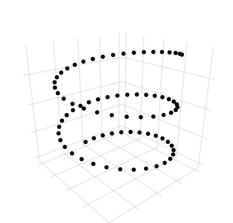
text(rich(string))



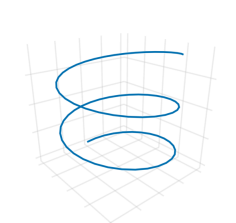
barplot(;color=pattern)



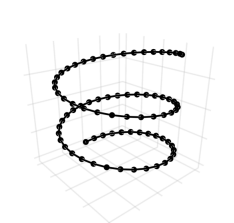
scatter(x,y,z)



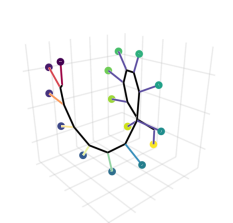
lines(x,y,z)



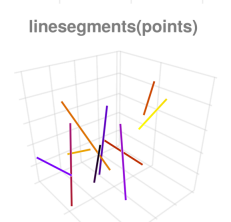
scatterlines(x,y,z)



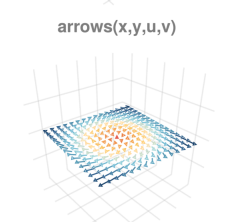
stem(x,y,z)



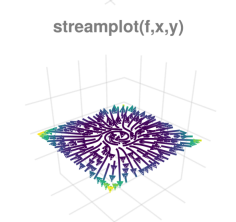
linesegments(points)



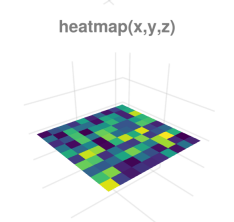
arrows(x,y,u,v)



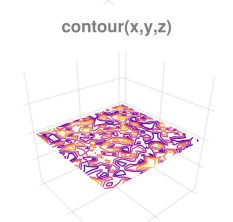
streamplot(f,x,y)



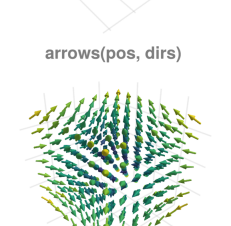
heatmap(x,y,z)



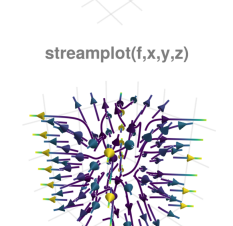
contour(x,y,z)



arrows(pos, dirs)

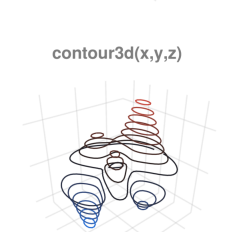


streamplot(f,x,y,z)

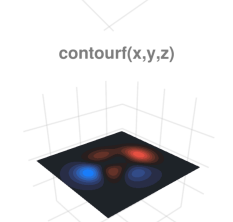


wireframe(x,y,z)

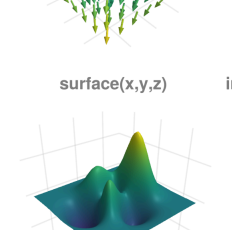
contour3d(x,y,z)



contourf(x,y,z)

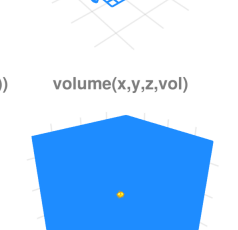


surface(x,y,z)

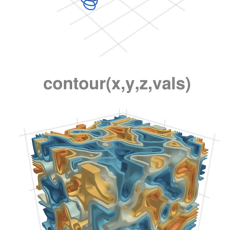


image(;transformation=(:yz,1))

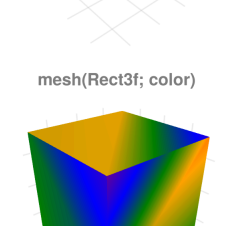
volume(x,y,z,vol)



contour(x,y,z,vals)

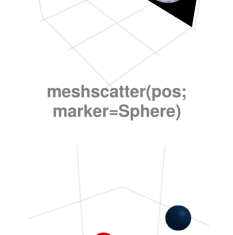


mesh(Rect3f; color)



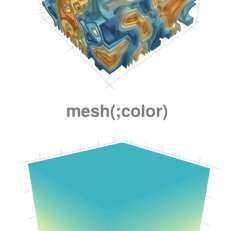
meshscatter(pos;
marker=Rect3f)

meshscatter(pos;
marker=Sphere)

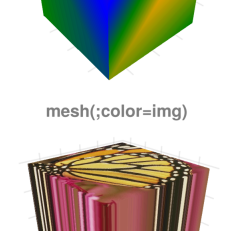


mesh(Sphere)

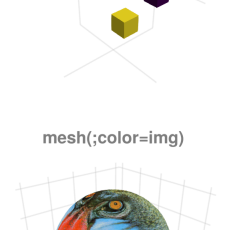
mesh(;color)



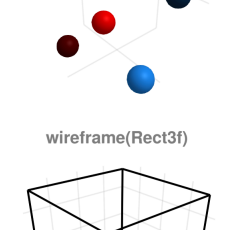
mesh(;color=img)



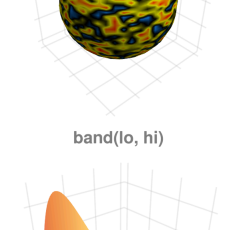
mesh(;color=img)



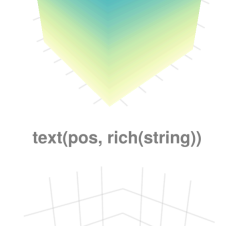
wireframe(Rect3f)



band(lo, hi)



text(pos, rich(string))



meshscatter(pos;
marker=Rect3f)