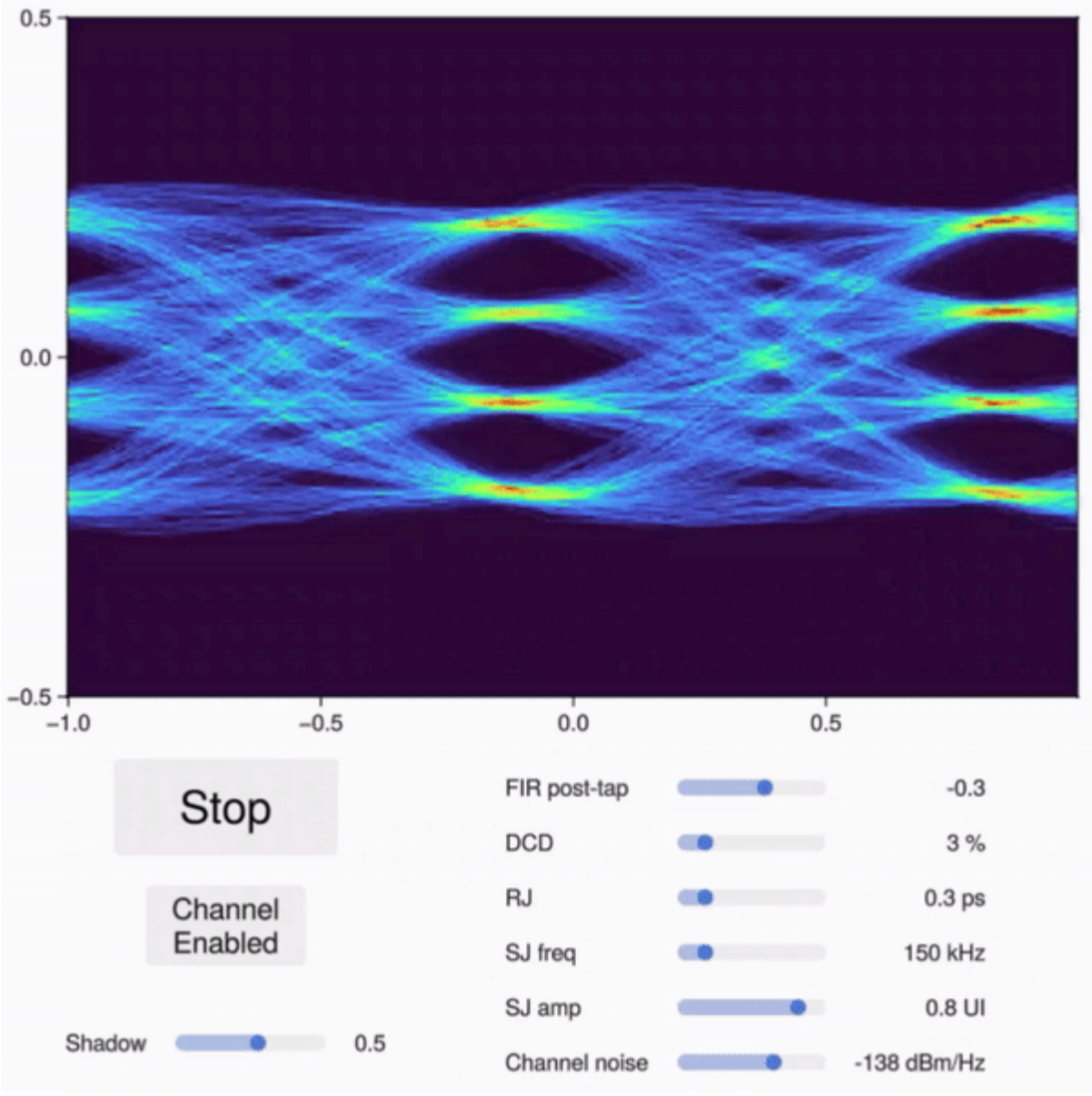
Building SerDes in Julia pt. 5B - Plotting w/ Animation



Finally we are here! This notebook will walk through how to build the animation and widget window shown above using Makie alone! The whole concept rests upon the idea of `Observables` - pointer like data structures that allow event listeners. Actually, you have already experienced the power of observables. This very Pluto notebook has observables in the background to achieve reactivity! Makie's own internal figure objects use observables too. Without further ado, let's begin building

So what are observables?

The idea of `Observables` is not that complicated. Just look at how simple the official documentation is compared to other Julia packages. The developers have put in quite some efforts to make Julia's observables easy to use and understand.

At a high level, an observable is like a pointer (or `Ref` in Julia) that contains the data of interest but can be watched with handlers. We create an observable variable like below

```
ob_name = Observable("Iron Man");
```

The argument to the `Observable()` constructor provides both an initial value and determines the type of the observable variable. So if I want to assign an interger to `ob_name`, it throw an error. There are two ways to access/update an observable's value - w/ `.val` or `[]`.

```
begin #these are two ways to access the data in an observable
println(ob_name)
println(ob_name.val);
println(ob_name[]);
end
```

```
Observable("Iron Man")
Iron Man
Iron Man
```

The difference is that only using `[]` will trigger the listner event when updating the observable value.

```
begin
ob_name.val = "Hulk" #no event is triggered, but value still updated
println(ob_name[])
end
```

```
Hulk
```

```
begin
ob_name[] = "Thor" #an update event will triggered and handler function will execute if defined
println(ob_name[])
end
```

```
Thor
```

```
MethodError: Cannot `convert` an object of type Int64 to an object of type String
Closest candidates are:
convert(::Type{String}, !Matched::Base.JuliaSyntax.Kind)
@ Base C:\workdir\base\JuliaSyntax\src\kinds.jl:975
convert(::Type{String}, !Matched::String)
@ Base essentials.jl:321
convert(::Type{String}, !Matched::FilePathsBase.AbstractPath)
@ FilePathsBase C:\Users\Kevin Z\.julia\packages\FilePathsBase\4RrDh\src\path.jl:117
...
```

Stack trace

Here is what happened, the most recent locations are first:

1. **setproperty!**(x::Observables.Observable{String}, f::Symbol, v::Int64) @ *Base.jl:40*
2. **setindex!**(observable::Observables.Observable, val::Any) @ *Observables.jl:122*
3. *This cell: line 1*

```
1 ob_name[] = 1 #ERROR, ob_name can only take String now
```

```
1 ob_name[] = 1 #ERROR, ob_name can only take String now
```

Event handlers are defined with the `on` function. Here is a simple example

```
1 begin
2
3 food_ob = Observable("pizza")
4 eat_func = on(food_ob) do food #every time the observable changes do the following
5     println(uppercasefirst(food) * " time!")
6 end
7
8 end;
```

```
1 food_ob[] = "coffee"; #try changing it here and see the handler function work
```

```
Coffee time!
```

```
1 food_ob.val = "Fried chicken"; #no printout if using .val to update, no matter how
  bad you want fried chicken
```

So how does this translate to Makie plots? Using observables, we only need to update the data and Makie will update the plot automatically! Let's build a simple plot

```
GLMakie.Screen(...)
```

```
begin
ftest = Figure();
axtest = Axis(ftest[1,1])
test_data = Observable(randn(10));
lines!(axtest, test_data);
ylims!(axtest,(-3.5, 3.5))
test_lbl = Label(ftest[2,1], "Rerun me!", tellwidth = false, fontsize=32);
display(GLMakie.Screen(), ftest); #run this cell to get a pop-up window
end
```

```
"Rerun me!"
```

```
begin #Rerun this cell to see the plot change by simply update test_data[]
test_lbl.text = "Running"
for n = 1:5
    test_data[] = randn(10)
    sleep(1)
end
test_lbl.text = "Rerun me!"
end
```

Congratualtions! We just created our first animation with Makie! Using observable completely separates the figure configuration and the actual data pipeline. Here we simply kept updating the observable data, and Makie replots immediately (Makie figure is essentially an event handler on the data itself).

A little rant before I move on: this "paradigm" is way better than what MATLAB has, which is a `drawnow` command after redrawing all the plots over and over again. I find myself spending more time on the plotting code than actually working on the model sometimes. Makie + Observable should save people a lot of time when data has to be constantly updated and visualized.

Chaining observables

Observables can also be chained together and each event can trigger more complicated processing down the line. We use `lift` to achieve this.

```
ob1 = Observable(2.0);
```

```
ob2 = lift((x)->x^2, ob1);
```

`ob2` is now a new observable that's chained with `ob1`, and we can create a event listener on `ob2`. Then, if we update `ob1`, the event handler on `ob2` will be triggered as well.

```
ob2_func = on(ob2) do val
    println(val-10);
end;
```

For more complicated chaining functions, we can either use `lift() do` or the `@lift` macro. Note w/ `@lift`, we can directly refer another observable with the `$` prefix.

```
ob2_v2 = lift(ob1) do val
    if val > 4
        return 4*val;
    else
        return val^2;
    end
end;
```

```
ob2_v3 = @lift begin
    return $ob1 > 4 ? 4*$ob1 : $ob1^2
end;
```

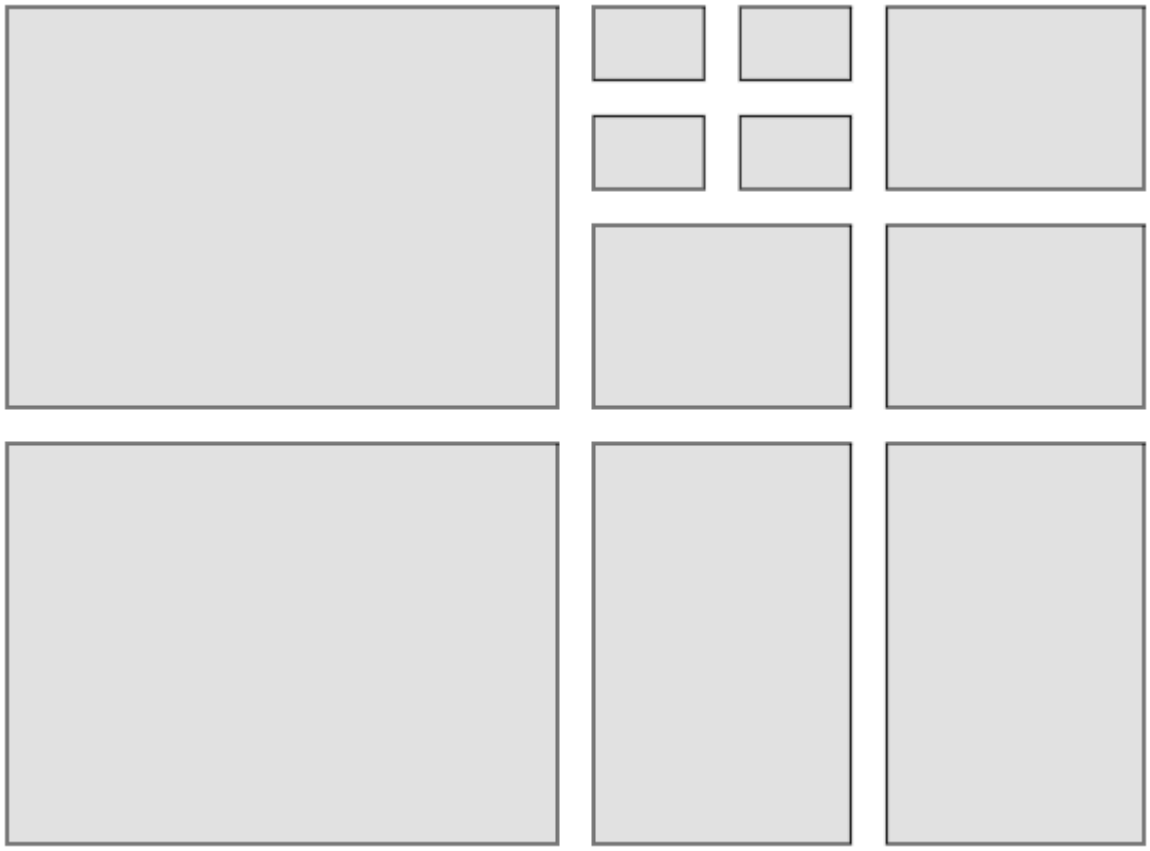
```
ob1[] = 2.0;
```

```
-6.0
16.0
16.0
```

Makie GridLayout and UI blocks

Remember in part 5a I complained about Makie's syntax not being suited for quick plotting and debug, and created my own `Figz` wrapper? Well, as it turned out Makie has some good reasons to have separate `GridLayout` and `Axis` systems. Makie supports many interactive UI elements directly in the figure window, which means the user might want to create complex layout like a GUI window. That's why not everything has an `Axis`, but they will all have associated grids.

You can arbitrarily create grids and subgrids with `[]` indexing on the figure object. This grid system allows us to do very flexible UI interface when it comes to both plots and controls. It's super easy in Makie to create grids with different sizes.

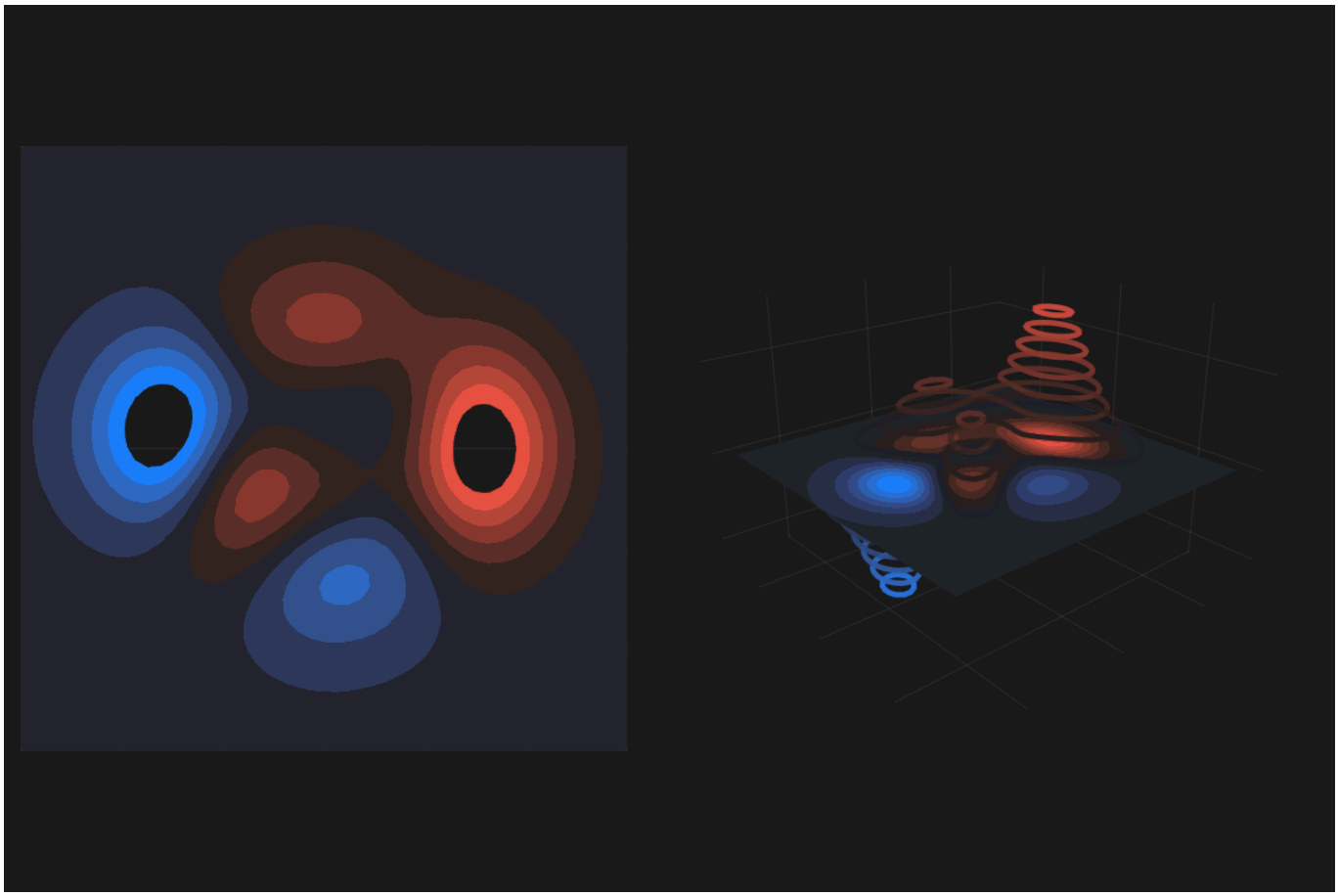


```
begin
fgrid = Figure();
#upper left
Box(fgrid[1,1])
#lower left
Box(fgrid[2,1])
#lower right into left and right
gridlr = GridLayout(fgrid[2,2]) #instead of keep indexing, a Gridlayout object can be created
Box(gridlr[1,1])
Box(gridlr[1,2])
#upper right into even smaller grids
gridur = GridLayout(fgrid[1,2])
sub_gridur = GridLayout(gridur[1,1])
Box(fgrid[1,2][1,1][1,1]) #either keep indexing
Box(gridur[1,1][1,2]) #or use grids/subgrids
Box(gridur[1,1][2,1])
Box(sub_gridur[2,2])
Box(gridur[1,2])
Box(gridur[2,1])
Box(gridur[2,2])
fgrid
end
```

Makie also provide built in basic interactive UI blocks, including `Button`, `Slider`, `Menu`, `Textbox`, and `Toggle`.


```
begin
fui = Figure(size=(500,100));

Label(fui[1,1], "Some text")
Button(fui[1,2], label="Click me")
Toggle(fui[1,3], active = true)
Slider(fui[1,4], range = 0:10, startvalue = 5, width = 100)
Menu(fui[1,5], options =["coffee", "tea", "more coffee"], default="coffee")

fui
end
```

These grant us enough flexibility and options to begin building the eye diagram widget! To learn more about Makie, go through their well written tutorials, and find more examples at BeatifulMakie, where you can copy/paste code and generate things like this right away:



It's alive!

Let's recap our simulation framework and generate some signals first before getting carried away with pretty plots.

Our simulation framework uses structs and mutating functions to model each circuit block's parameters and operations. In this tutorial, we will use the TX model (details in part 4) and add a toy channel. The channel will include both a PCB trace (translated to impulse response from .s4p frequency response) and another 1st order pad response. The additive Gaussian noise level is specified in dBm/Hz.

```
begin
param = TrxStruct.Param(
            data_rate = 56e9,
            pam = 4,
            osr = 20,
            blk_size = 2^10,
            subblk_size = 32,
            nsym_total = Int(1e6));

bist = TrxStruct.Bist(
            param = param,
            polynomial = [28,31]);

drv = TrxStruct.Drv(
        param = param,
        ir = u_gen_ir_rc(param.dt, param.fbaud/2, 20*param.tui), #1st order ir
        fir = [1.0, -0.0],
        swing = 0.8,
        jitter_en = true,
        dcd = 0.00,
        rj_s = 000e-15,
        sj_amp_ui = .0,
        sj_freq = 2e5);

ch = TrxStruct.Ch(
            param = param,
            ir_ch = u_fr_to_imp("../../channel_data/TF_data/channel_4inch.mat",
                    param.tui, param.osr, npre = 20, npost= 79),
            ir_pad = u_gen_ir_rc(param.dt, param.fbaud, 20*param.tui),
            noise_en = true,
            noise_dbm_hz = -140)

trx = (;param, bist, drv, ch);
end;
```

Now that our TX + channel isntances are instantiated, we will create a `step_sim_blk` function, which basically runs one block of the simulation and we will use the generated waveform to create one

frame of the eye diagram. Note that this `step_sim_blk` function is taking an `ch_en` argument which allows us to toggle between channel included or excluded.

At a high level, if block size is too small, the frame rate will naturally be faster but noisier per frame. If the block size is too big, frame rate drops but each frame gets more data for plot. ~1000 symbols per block seem to be a good tradeoff. Feel free to play around with these paramters. To make it more fun, we will do a PAM4 eye this time.

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

```
function step_sim_blk(trx; ch_en)
    @unpack bist, drv, ch = trx

    pam_gen_top!(bist)

    dac_drv_top!(drv, bist.So)

    if ch_en
        ch_top!(ch, drv.Vo)
        return copy(ch.Vo)
    else
        return copy(drv.Vo)
        #we are returning copies of drv/ch.Vo because remember they are
        views/subarrays
    end

end
```

If you followed my previous notebooks, the following code blocks should look familiar and you tweak the eye diagram parameters below.

```
begin
x_nui = 2
x_npts_ui = 256
x_npts = x_nui*x_npts_ui
y_range = 1 #+/-0.5
y_npts = 256
x_grid = -x_nui/2: 1/x_npts_ui:x_nui/2-1/x_npts_ui
y_grid = -y_range*(0.5-.5/y_npts):y_range/y_npts:y_range*(0.5-.5/y_npts)
end;
```

Now we will use observable to define the eye diagram to be plotted. `eye_buffer` will observe the time domain waveform of either driver or channel output (again, the initial value doesn't really matter, but the type needs to be defined). `last_heatmap` and `shadow_weight` are used for creating "after images" of eye diagrams - think of it as a shadowing that slowly decays.

```
eye_buffer = Observable(copy(drv.Vo));
```

```
last_heatmap = zeros(x_npts,y_npts);
```

```
shadow_weight = Observable(0.5);
```

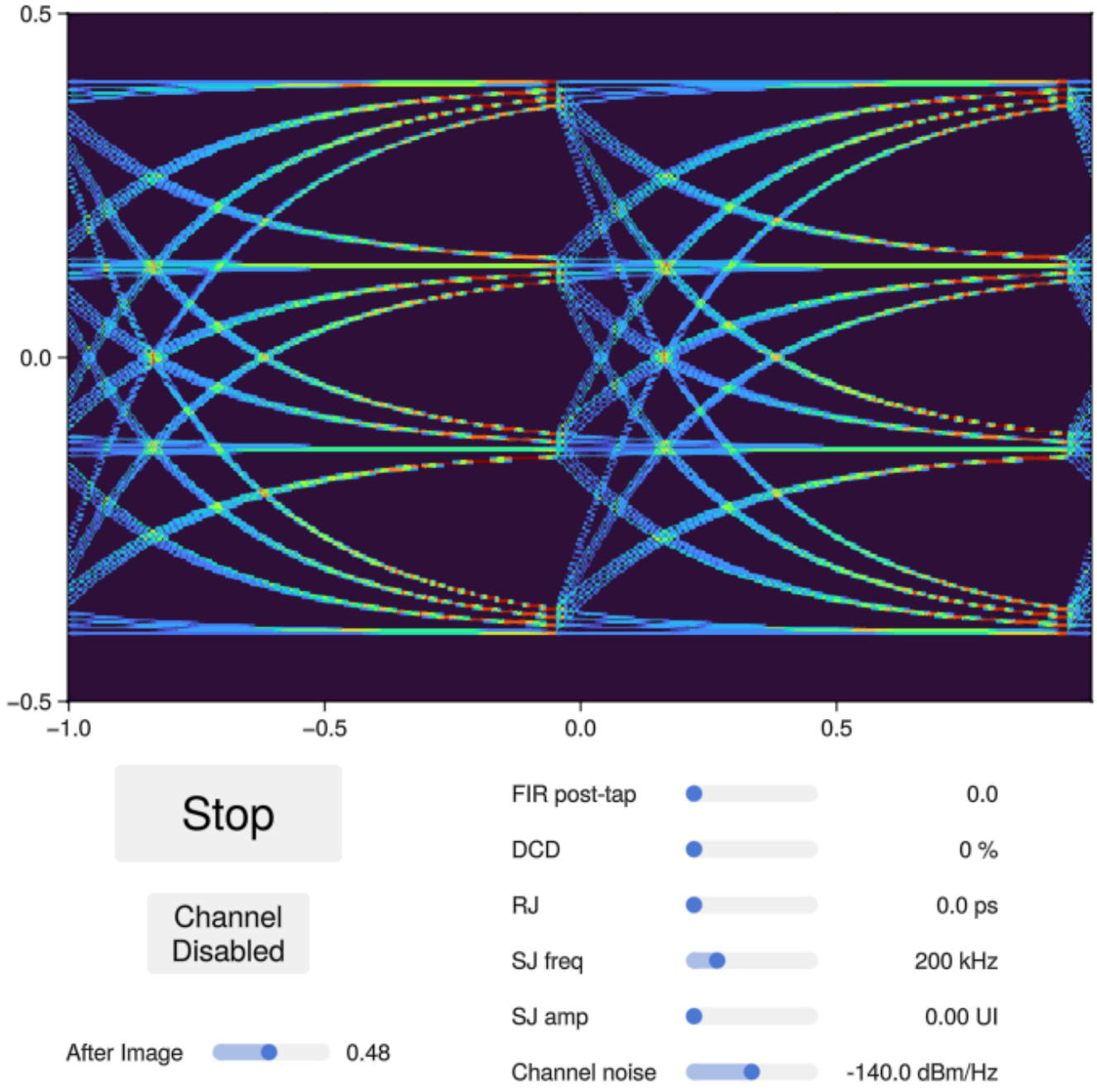
Here we create a new observable, chained from `eye_buffer`. So eveytime `eye_buffer` is updated, this new observable `eye_heatmap` will be update automatically. We will be fancy here and put the eye diagram through a leaky integrator (i.e. `eye_out[n] = eye_in[n] + α × eye_out[n-1]`, `α∈[0,1)`) to create the after image effect.

```
eye_heatmap = @lift begin
    cur_heatmap = shadow_weight[]*last_heatmap .+ w_gen_eye_simple($eye_buffer,
    x_npts_ui, x_npts, y_range, y_npts; osr = trx.param.osr)
    last_heatmap = cur_heatmap
    cur_heatmap #returns the current heatmap
end;
```

Time to make the widget UI! The entire figure creation is capture in the following code block. The widget includes a run button to start/stop the simulation, a button for enabling/disabling channel (i.e. ISI). A set of sliders for tuning TX/channel parameters real time. And a small slider for setting the after image weight.

```
begin #run this block to see the widget!
f = Figure(size=(700,700))
#plot the eye diagram, pass the observable directly in
heatmap!(Axis(f[1:2, 1:3]), x_grid, y_grid, eye_heatmap, colormap=:turbo)

#run button
btn_run = Button(f[3,1][1,1], label ="Run", tellwidth=false, tellheight=false, width
= 140, height=60, fontsize=28);
#enable channel button
btn_ch_en = Button(f[3,1][2,1], label="Channel\nDisabled", tellwidth = false,
tellheight=false, width=100, height=50, fontsize=18);
#slider for after image weight
sl_shadow = SliderGrid(f[3,1][3,1], (label="After Image",  range=0:0.01:1,
startvalue=shadow_weight[]), tellwidth = false, tellheight=false, width=200);
#sliders for TX/Channel parameters
sl = SliderGrid(f[3,2:3],
    (label = "FIR post-tap", range=0:-0.05:-0.5, startvalue = trx.drv.fir[2]),
    (label = "DCD", range=0:1:20, format = "{:d} %", startvalue = trx.drv.dcd*100),
    (label = "RJ", range=0:0.1:2, format = "{:.1f} ps", startvalue = trx.drv.rj_s/1e-
12),
    (label = "SJ freq", range=0:50:1000, format = "{:d} kHz", startvalue =
    trx.drv.sj_freq/1e3),
    (label = "SJ amp", range = 0:0.05:1, format = "{:.2f} UI", startvalue =
trx.drv.sj_amp_ui),
    (label = "Channel noise", range=-150:.5:-130, format = "{:.1f} dBm/Hz",
    startvalue = trx.ch.noise_dbm_hz),
    tellheight=false, width = 300)
#open a new figure window
display(GLMakie.Screen(), f); #uncomment and see the widget in a pop-up window
end;
```

When you run the code block above, a window should open up. After you hit run, you should see sometime like this



The section below contains the code related to how each UI element works. Each one is simple enough and should be self explanatory

```
begin
ch_en = Observable(false); #this is passed to the step_sim_blk function above
ch_en_func = on(btn_ch_en.clicks) do clicks
    ch_en[] = ~ch_en[];
    btn_ch_en.label = ch_en[] ? "Channel\nEnabled" : "Channel\nDisabled"
end
end;
```

```
begin
eye_shadow = lift(sl_shadow.sliders[1].value) do val
    shadow_weight[] = (val == 1.0) ? 0.9999 : val #when weight =1, we purposely make
    it very close 1 becausue a perfect integrator can cause Float overflow
end
end;
```

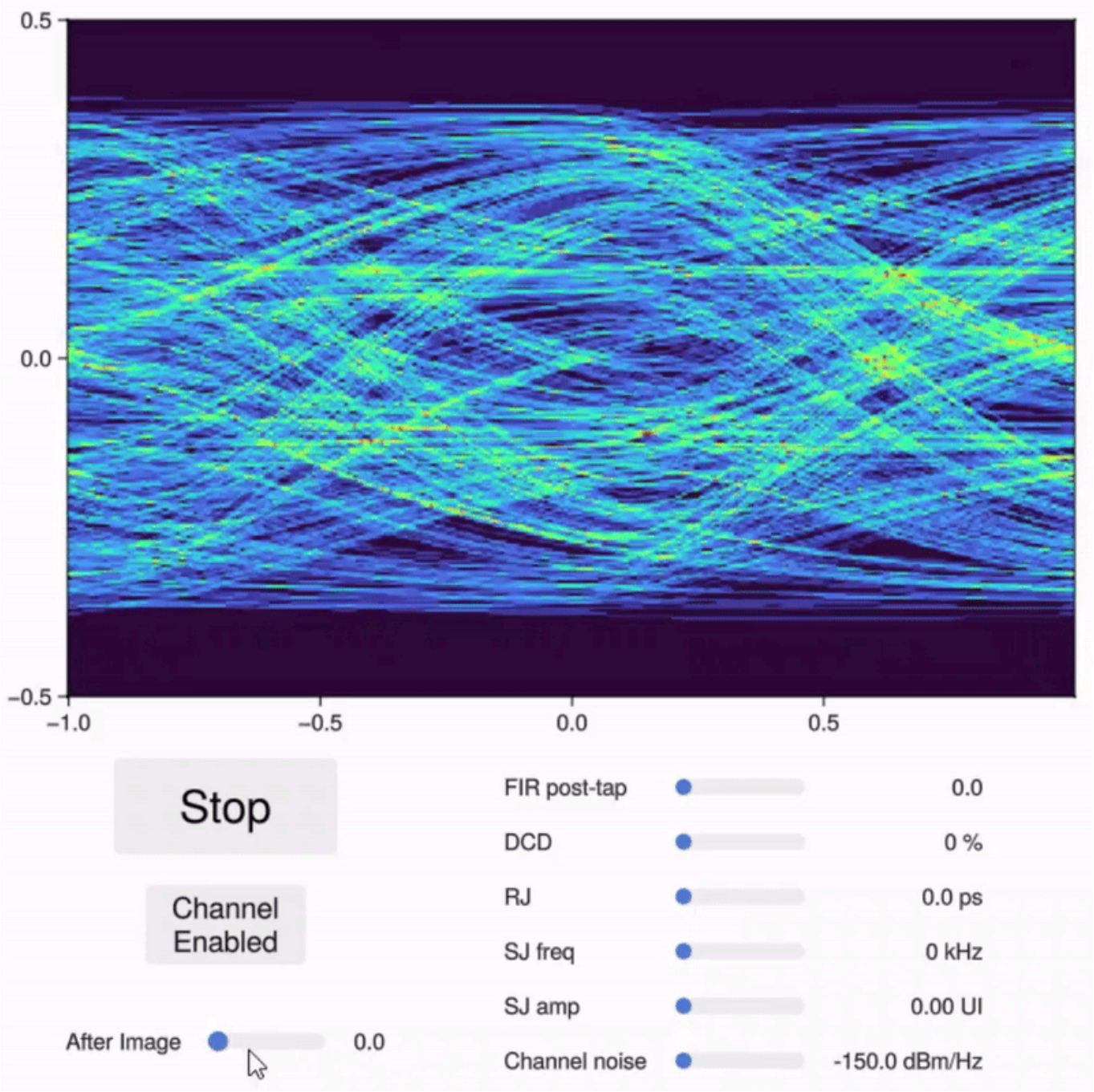
```
begin
sliderobservables = [s.value for s in sl.sliders]
trx_settings = lift(sliderobservables...) do slvalues...
    trx.drv.fir[2] = slvalues[1]
    trx.drv.dcd = slvalues[2]/100
    trx.drv.rj_s = slvalues[3]*1e-12
    trx.drv.sj_freq = slvalues[4]*1e3
    trx.drv.sj_amp_ui = slvalues[5]
    trx.ch.noise_dbm_hz = slvalues[6]

    trx.drv.fir_norm = trx.drv.fir/sum(abs.(trx.drv.fir))
    trx.ch.noise_rms = sqrt(0.5/trx.param.dt*10^((trx.ch.noise_dbm_hz-
    30)/10)*trx.ch.noise_Z) #explicitly recalculate noise rms because it's not an
    observable
end
end;
```

The most important UI element is the run button and this is similar to the one shown in the tutorial here. Our animation step/frame is simply done by updating the `eye_buffer` observable, which in turn triggers the `eye_heatmap` generation, then Makie will update the plot. I wish I can say more about this, but it's really just this simple to make it work!

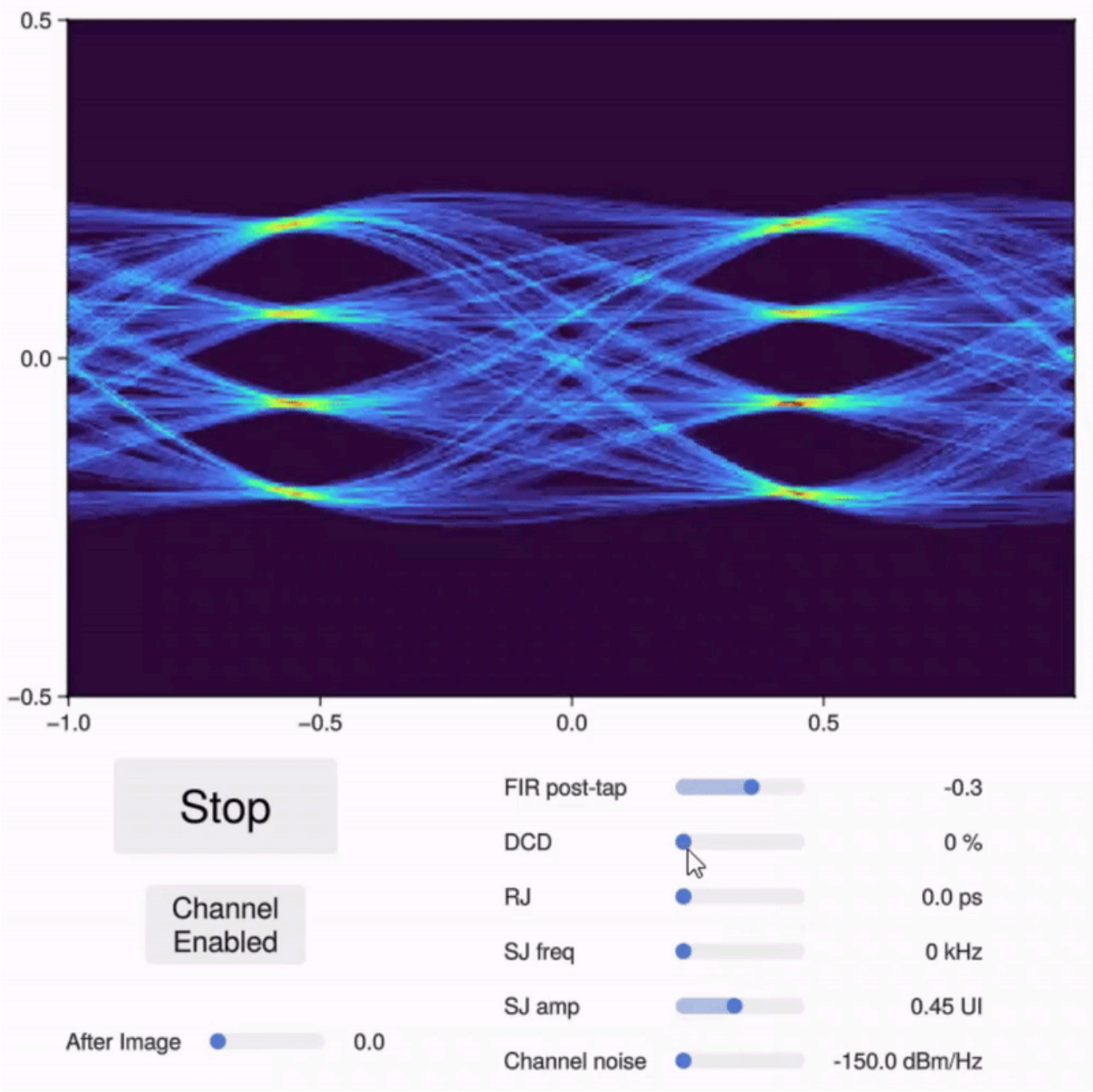
```
begin
isrunning = Observable(false);

run_func1 = on(btn_run.clicks) do clicks
    isrunning[] = ~isrunning[];
    btn_run.label = isrunning[] ? "Stop" : "Run"
end;

run_func2 = on(btn_run.clicks) do clicks
    @async while isrunning[]
        isopen(f.scene) || break
        eye_buffer[] = step_sim_blk(trx, ch_en=ch_en[]); #this is the animation step
        yield()
        # sleep(0.001) #use sleep to slow down the frame rate if desired
    end
end
end;
```

Now that we have reached the end, I will conclude not with too many words, but GIFs of what this widget can generate. **Enjoy!** 👏

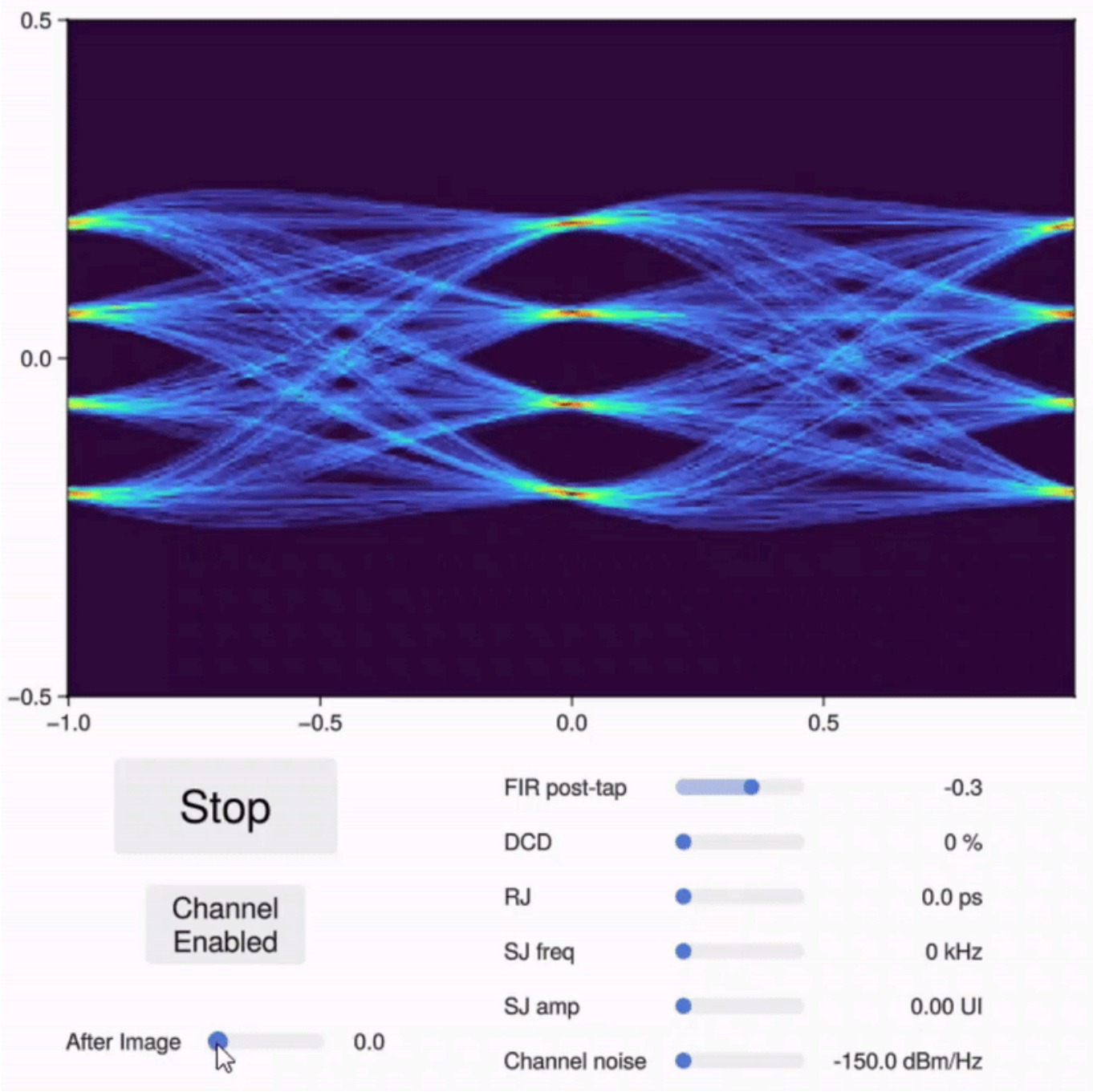
ISI


0.5
0.0
−0.5
−1.0
−0.5
0.0
0.5
Stop
Channel Enabled
After Image
0.0
FIR post-tap
0.0
DCD
0 %
RJ
0.0 ps
SJ freq
0 kHz
SJ amp
0.00 UI
Channel noise
-150.0 dBm/Hz

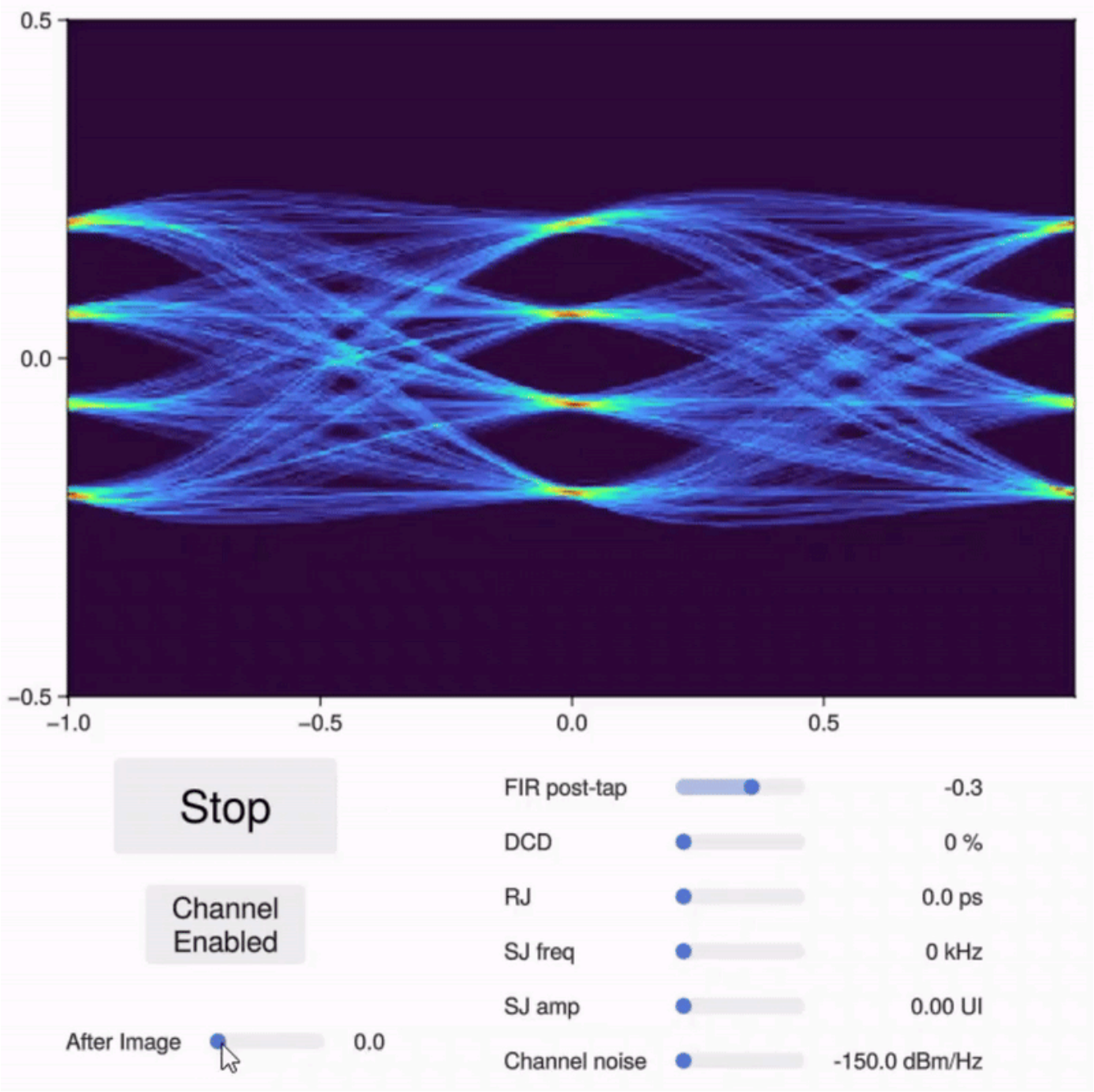
DCD


0.5
0.0
−0.5
−1.0
−0.5
0.0
0.5
Stop
Channel Enabled
After Image
0.0
FIR post-tap
-0.3
DCD
0 %
RJ
0.0 ps
SJ freq
0 kHz
SJ amp
0.45 UI
Channel noise
-150.0 dBm/Hz

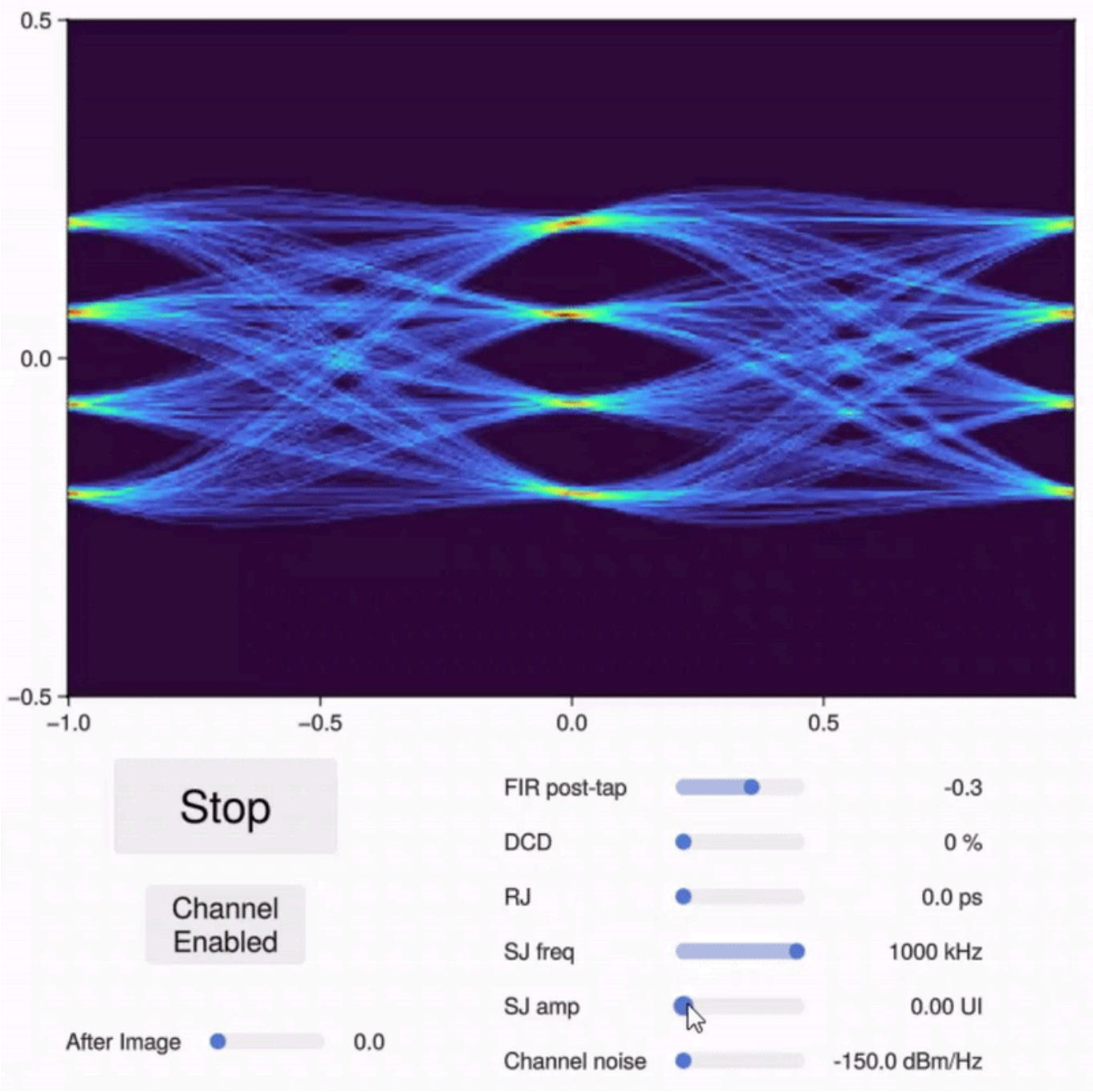
Noise


0.5
0.0
−0.5
−1.0
−0.5
0.0
0.5
Stop
Channel Enabled
After Image
0.0
FIR post-tap
-0.3
DCD
0 %
RJ
0.0 ps
SJ freq
0 kHz
SJ amp
0.00 UI
Channel noise
-150.0 dBm/Hz

RJ


0.5
0.0
−0.5
−1.0
−0.5
0.0
0.5
Stop
Channel Enabled
After Image
0.0
FIR post-tap
-0.3
DCD
0 %
RJ
0.0 ps
SJ freq
0 kHz
SJ amp
0.00 UI
Channel noise
-150.0 dBm/Hz

SJ



Helper and import section

```
using Parameters, UnPack, DSP, FFTW, Random, Interpolations, DataStructures, Distributions, StatsBase, MAT
```

```
using GLMakie, Makie
```

```
include("../../src/structs/TrxStruct.jl");
```

```
begin
include("../../src/blks/BlkBIST.jl");
import .BlkBIST: pam_gen_top!
end
```

```
begin
include("../../src/blks/BlkTX.jl");
import .BlkTX: dac_drv_top!
end
```

```
begin
include("../../src/blks/BlkCH.jl");
import .BlkCH: ch_top!
end
```

```
begin
include("../../src/util/Util_JLSD.jl");
import .Util_JLSD: u_gen_ir_rc, u_fr_to_imp
end
```

```
begin
include("../../src/blks/WvfmGen.jl");
import .WvfmGen: w_gen_eye_simple
end
```