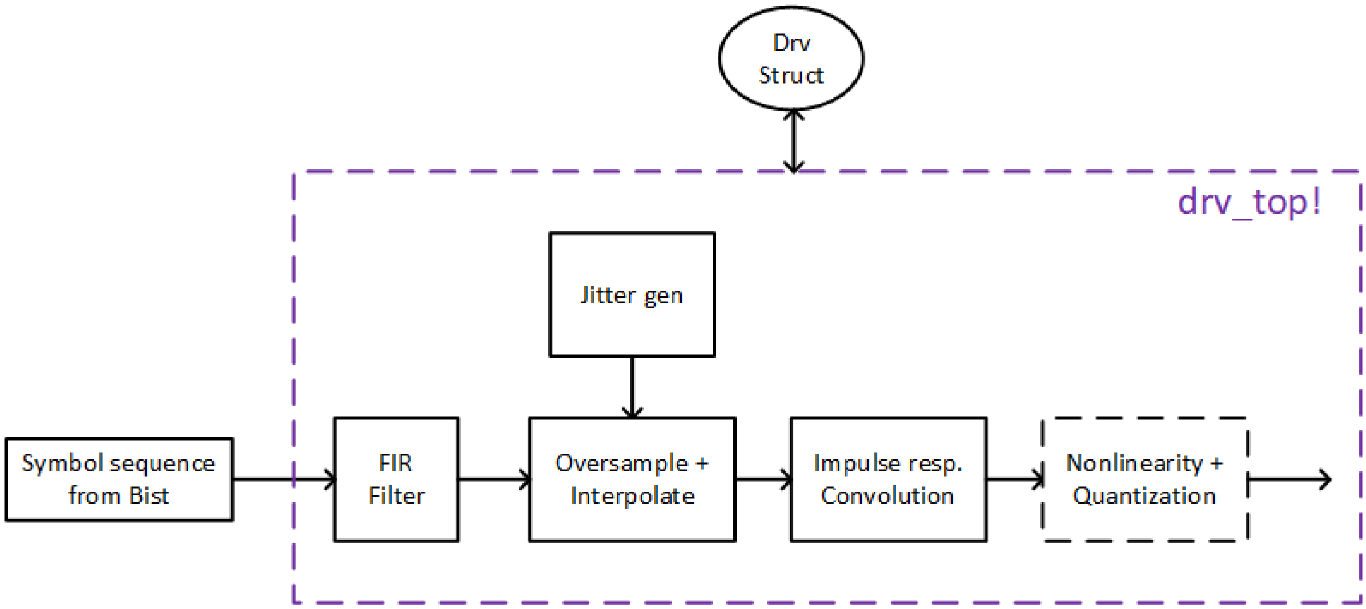
Building SerDes Models in Julia, pt4 - Detailed Transmitter Example



Let's build some meaningful models! We will use a transmitter as an example to show some important concepts in this simulation framework and Julia.

To set the stage better, this notebook will have codes both in a cell (which you can run), and shown as just text. We have included/imported relevant source files into the notebook. The text block is just showing what the source code is to avoid variable definition collision, like below. The actual `Drv` struct is instantiated from the `TrxStruct` module.

```julia
@kwdef mutable struct Drv
    const param::Param

    ir::Vector{Float64}
    swing = 0.7

    fir::Vector{Float64} = [1,0]
    fir_norm = fir/sum(abs.(fir))

    #not used in the current model
    rlm_en = false
    rlm = 1.0
    quantize = false
    dac_res = 7

    jitter_en = false
    dcd = 0.0
    rj_s = 0.0
    sj_amp_ui = 0.0
    sj_freq = 0.0
    last_sj_phi = 0.0

    Sfir_conv::Vector = zeros(param.blk_size+length(fir)-1)
    Sfir = @views Sfir_conv[1:param.blk_size]
    Sfir_mem = @views Sfir_conv[param.blk_size+1:end]
    Vfir = zeros(param.blk_size_osr)

    prev_nui = 4
    Δtt_ext = zeros(prev_nui+param.blk_size+1)
    Δtt = zeros(param.blk_size)
    Δtt_prev_nui = @views Δtt_ext[end-prev_nui:end]
    Vext::Vector = zeros(prev_nui*param.osr+param.blk_size_osr)
    V_prev_nui = @views Vext[end-prev_nui*param.osr+1:end]
    tt_Vext::Vector = zeros(prev_nui*param.osr+param.blk_size_osr)
    tt_uniform::Vector = (0:param.blk_size_osr-1) .+ prev_nui/2*param.osr

    Vo_conv::Vector = zeros(param.blk_size_osr+lastindex(ir)-1)
    Vo = @views Vo_conv[1:param.blk_size_osr]
    Vo_mem = @views Vo_conv[param.blk_size_osr+1:end]

end
```

What's in the TX model?

Let's go through what's included in the transmitter/driver before discussing how each portion is modeled.

```julia
    ir::Vector{Float64}
    swing = 0.7
```

First, the driver will have its own continuous-time impulse response to model its bandwidth. Here a time-domain vector `ir` is used since we can also use SPICE simulated impulse response if desired. The `ir` vector has to use the simulation time step in `param` (i.e., use the same `osr` per symbol). This can be done during initialization. `swing` specifies the peak-to-peak magnitude of the driver's output signal.

```
Vo_conv::Vector = zeros(param.blk_size_osr+lastindex(ir)-1)
Vo = @views Vo_conv[1:param.blk_size_osr]
Vo_mem = @views Vo_conv[param.blk_size_osr+1:end]
```

The `Vo_*` vectors are used for storing the TX's output waveforms and will be passed to the next block. Here we introduce a `"view"` of an array. A `view` is essentially a pointer to a sub-section of another vector, but not a standalone vector itself. For example:

```
begin
one2ten = collect(1:10); #this is to convert UnitRange to Array

#this creates a copy of one2ten[1:5] and assign it to one2five
one2five = one2ten[1:5];

#this says one2five_view points to the section 1:5 of one2ten, but it's not an independent vector
one2five_view = view(one2ten, 1:5);
end;
```

If now we change the first element of `one2ten`, `one2five` will not change, but `one2five_view` will because it's pointing to a sub-array `one2ten`

```
begin
one2ten[1] = 10; #change the first element here to see how the other two changes

println(one2five[1])
println(one2five_view[1])
end
```

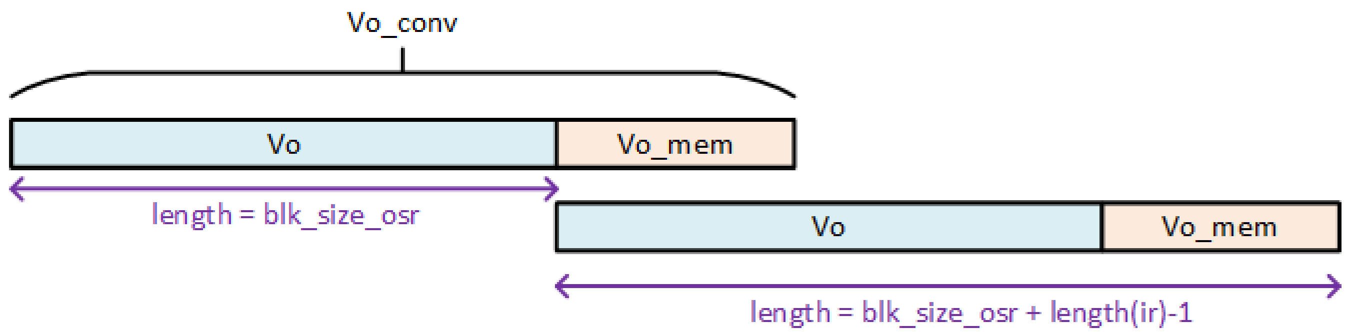
```
1
10
```

Julia Tips

`@views` is a macro that converts sliced arrays into views (pointers are much cheaper than creating copies of arrays). For more information on how to use the `view` syntax correctly, check here

With views, we can declare only one memory space for `Vo_conv`, and create convience variables to make the first `blk_size_osr` length vector the actual output vector, and the rest becomes the memory vector that overalps with the beginning of the next block. The length for `Vo_conv` is known beforehand given the length of the `ir` vector and `blk_size_osr`.

This way, there is no need to explicitly copy and define `Vo = Vo_conv[1:blk_size_osr]` during our simulation to save space and time (something that's not possible in MATLAB). We will revisit this again when writing the convolution function later.




```
fir::Vector{Float64} = [1,0]
fir_norm = fir/sum(abs.(fir))
```

```
Sfir_conv::Vector = zeros(param.blk_size+length(fir)-1)
Sfir = @views Sfir_conv[1:param.blk_size]
Sfir_mem = @views Sfir_conv[param.blk_size+1:end]
Vfir = zeros(param.blk_size_osr)
```

`fir` is a small vector containing the discrete-time coefficients for TX de-emphasis. `fir_norm` is the normalized coefficients to model the peak power constraint. Here we will just calculate it internally during initialization. Going along with `fir` are the internal vectors used for convolution, `Sfir_conv`. Similarly, `Sfir` and `Sfir_mem` are the sub-array views. `Vfir` is just the oversampled version (by `osr` times) of the filtered sequence `Sfir`.

```
jitter_en = false
dcd = 0.0              #0.01 = 1%
rj_s = 0.0             #random jitter in seconds
sj_amp_ui = 0.0        #sinusoidal jitter amplitude in UI
sj_freq = 0.0          #sinusoidal jitter frequency in Hertz
last_sj_phi = 0.0      #state variable for previous blk's last SJ phase
```

```
prev_nui = 4
Δtt_ext = zeros(prev_nui+param.blk_size+1)
Δtt = zeros(param.blk_size)
Δtt_prev_nui = @views Δtt_ext[end-prev_nui:end]
Vext::Vector = zeros(prev_nui*param.osr+param.blk_size_osr)
V_prev_nui = @views Vext[end-prev_nui*param.osr+1:end]
tt_Vext::Vector = zeros(prev_nui*param.osr+param.blk_size_osr)
tt_uniform::Vector = (0:param.blk_size_osr-1) .+ prev_nui/2*param.osr
```

These parameters/variables are used for modeling TX jitter. Currently the model supports duty cycle distortion (`dcd`), normally distributed random jitter (`rj`) and sinuosoidal jitter (`sj`). The parameters are defined in their "most comfortable" units.

`prev_nui` denotes the number of previous symbols to be stitched to the current block's signal to prevent overflow/underflow when jitter is introduced. `Δtt*` vectors store the jitter information at each edge location. The `tt_Vext` vector is the jittered time grid vector. `tt_uniform` is the convience vector to remap the jittered waveform back to our simulation grid. `Vext` and `V_prev_nui` are the extended block vectors and the previous N-UI symbols. More details on how these are used to model jitter later.

```
#not used in the current model
rlm_en = false
rlm = 1.0
quantize = false
dac_res = 7
```

What's also possible is to model nonlinearity and quantization errors in the TX driver (if a DAC based model is desired). In fact, it would be a good exercise to do to extend this model and learn Julia on your own 😃.

Ok, time to extend the relevant structs

```
#run this cell if you don't see any waveforms below
begin
param = TrxStruct.Param(
            data_rate = 10e9,
            pam = 2,
            osr = 32,
            blk_size = 2^14,
            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/4, 20*param.tui), #1st order ir
        fir = [1.0, -0.2],
        swing = 0.8,
        jitter_en = true,
        dcd = 0.03,
        rj_s = 300e-15,
        sj_amp_ui = 0.0,
        sj_freq = 10e6);
end;
```

The `drv_top!` function

Let's begin with some pseudo-code according to the block diagram at the very top. The `drv_top!` function would take the `drv` struct as an input (which will contain all the necessary parameters, internal states and pre-allocated output vectors), and a bit sequence vector from the BIST block.

```
function drv_top!(drv, input)
    @unpack all parameters and vectors

    apply_fir_filter!(Sfir, input, fir, kwargs...)

    oversample!(Vfir,Sfir)

    if jitter_en
        add_jitter!(drv, Vfir)
    end

    convolve!(Vo_conv, Vfir, ir, kwargs...)

end
```

We will begin with the convolution function since it can belong to the utility module and called by many other circuit blocks (anything with an impulse response really).

```
u_conv! (generic function with 1 method)
```

```
function u_conv!(Vo_conv, input, ir; Vi_mem = Float64[], gain = 1)
    Vo_conv[eachindex(Vi_mem)] .= Vi_mem
    Vo_conv[lastindex(Vi_mem)+1:end] .= zero(Float64)

    Vo_conv .+= conv(gain .* input, ir)
    return nothing
end
#we will use a "u_" prefix for all utility functions to avoide name collision in the future
```

A quick recap: the `!` is a Julian *convention* that denotes the function will mutate one or more of the arguments (usually the first argument). Here we pass in the pre-allocated output vector, `Vo_conv`, `input` and `ir`. Optional arguments include a memory vector (from the previous block) `Vi_mem`, and a `gain` factor. The gain factor comes in handy for general normalization (i.e. multiply by `dt` in convolution).

Note that I didn't need to assign the specific sub-arrays of `Vo_conv` to `Vo` and `Vo_mem`. This is automatically maintained with `views`. We also directly use the `conv` function from DSP.jl since it's quite optimized with FFT. For continuous-time convolutions, the input and ir vectors could be pretty long, so FFT-based convolution is more suitable.

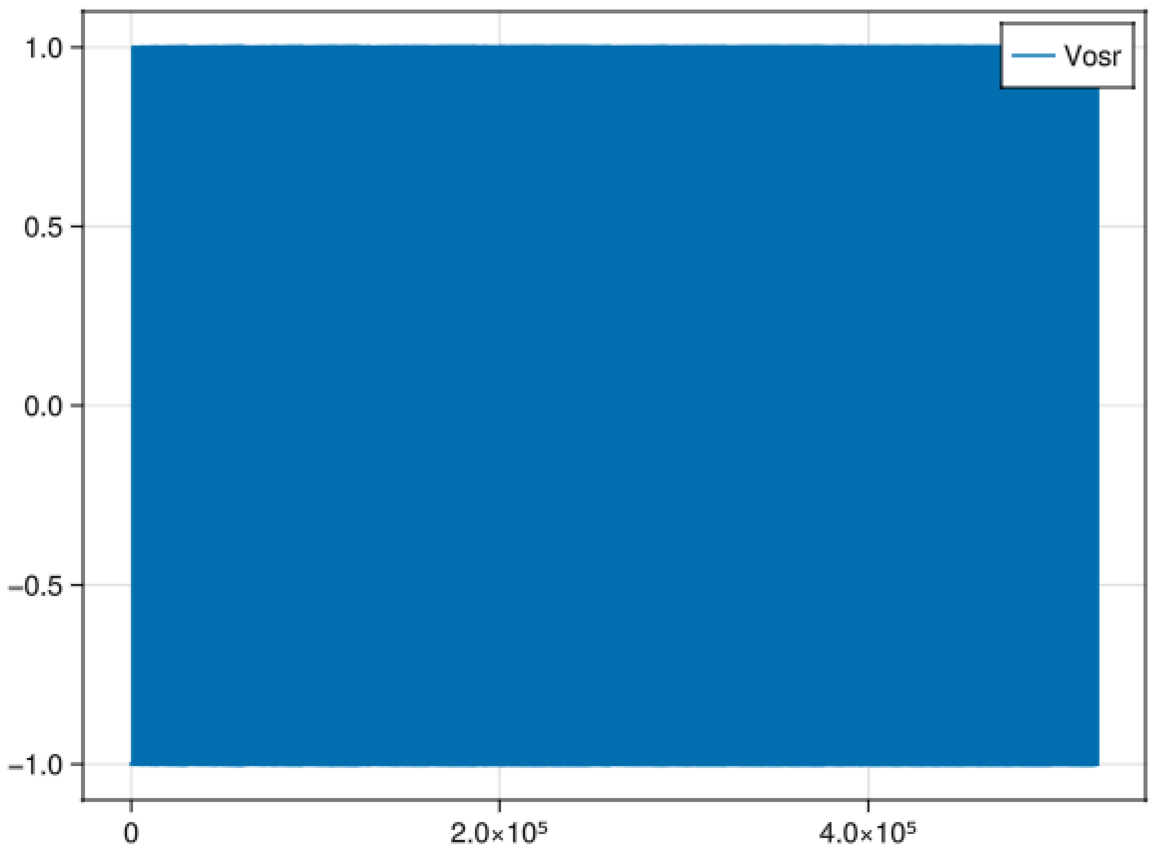
```
u_conv (generic function with 1 method)
```

```julia
#we will also create a non-mutating u_conv function for other uses
function u_conv(input, ir; Vi_mem = zeros(1), gain = 1)
    vconv = gain .* conv(ir, input)
    vconv[eachindex(Vi_mem)] += Vi_mem

    return vconv
end
```

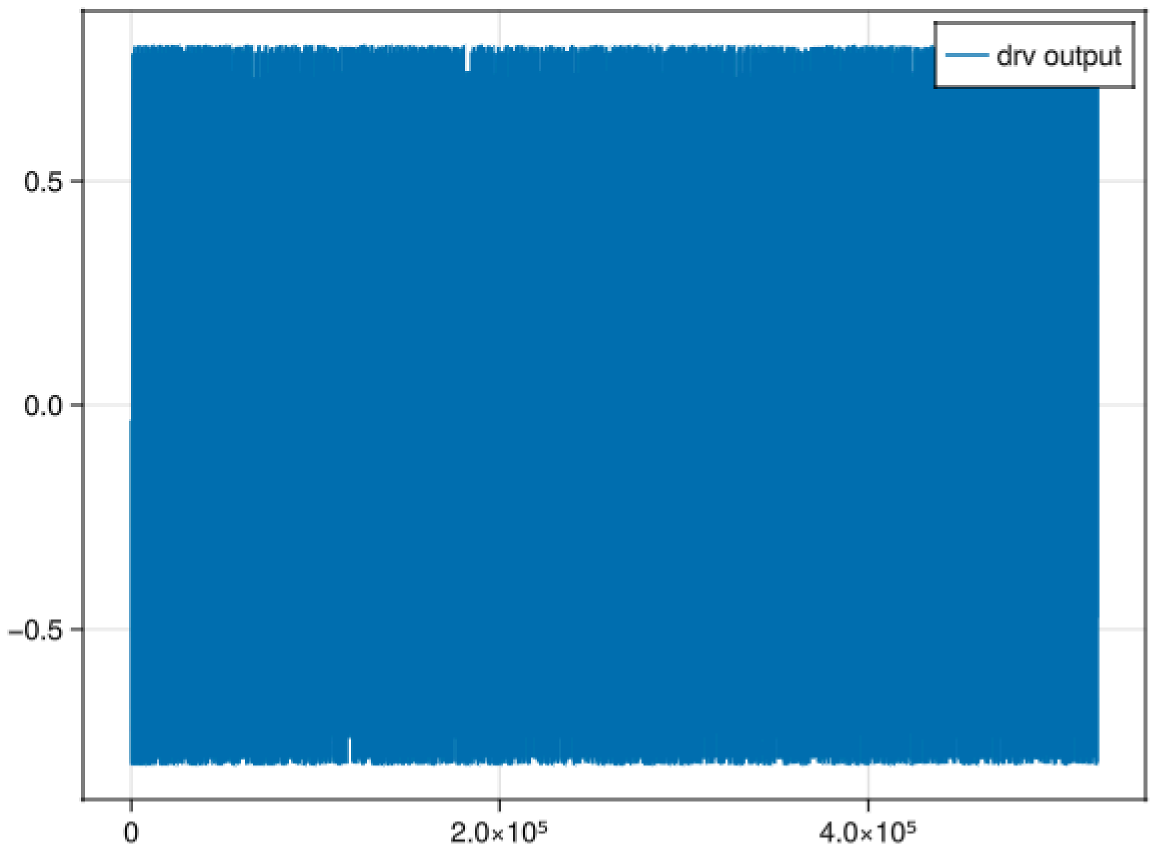
Let's test it out:

```julia
pam_gen_top!(bist) #generate some PRBS bits
```

```julia
Vosr = kron(bist.So, ones(param.osr));
#oh yeah, oversampling is this simple too
```



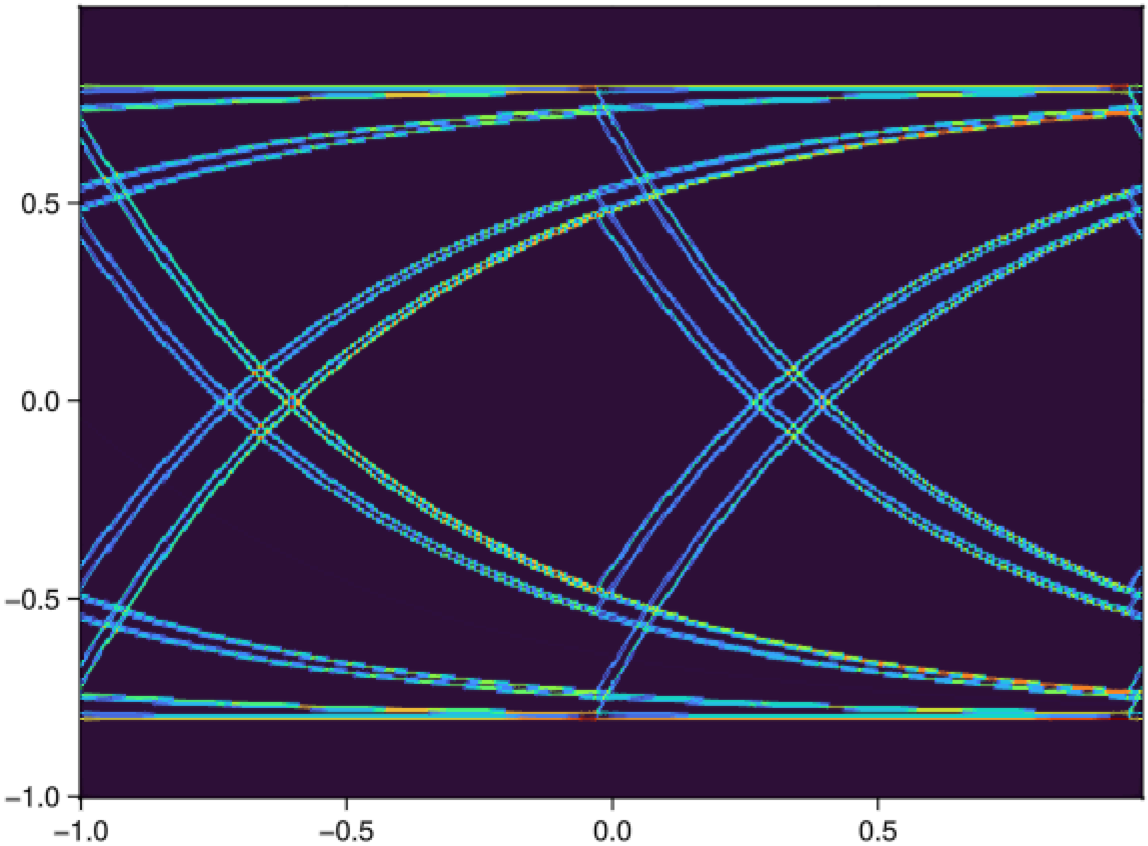

```julia
#call our convolution function; let's keep the input memory zero for now
#change the drv parameters in the struct definition to see the waveform/eye change
u_conv!(drv.Vo_conv, Vosr, drv.ir, Vi_mem=zeros(1), gain = drv.swing * param.dt);
```



For those who want to see eye diagrams, a helper eye diagram generation (heatmap based) function is included and we can plot the TX output like below (make sure to increase `blk_size` above to get more samples. Re-run cells if no change is seen). If the TX waveform is pretty "clean", the generated eye diagram might be a bit hard to see.

```
begin #common eye diagram params
x_npts_ui = 256; #number of points per ui
x_nui = 2;
x_npts = x_nui*x_npts_ui; #plot two UI eye diagram
x_grid = -x_nui/2 : 1/x_npts_ui: x_nui/2-1/x_npts_ui;
y_range = 2;
y_npts = 256; #number of points on y axis
y_grid = -y_range/2: y_range/y_npts: y_range/2 - y_range/y_npts;
end;
```

```
eye_tx = w_gen_eye_simple(drv.Vo,
                            x_npts_ui, x_npts,
                            y_range, y_npts,
                            osr= param.osr, x_ofst=0);
```



That wasn't too bad, was it? Actually, we can directly use the `u_conv!` function for applying the FIR filter as well. However, the FIR filter typically is much shorter (<10 taps) than the symbol vector, using FFT convolution might be an overkill. For optimization, a simple shift-and-add filter function can be written as below

```
u_filt! (generic function with 1 method)
```

```
function u_filt!(So_conv, input, fir; Si_mem=Float64[])
    So_conv[eachindex(Si_mem)] .= Si_mem
    So_conv[lastindex(Si_mem)+1:end] .= zero(Float64)
    s_in = lastindex(input)

    for n=eachindex(fir)
        So_conv[n:s_in+n-1] .+= fir[n] .* input
    end

    return nothing
end
```

u_filt (generic function with 1 method)

```
#non-mutating version
function u_filt(input, fir; Si_mem=Float64[])
    sconv = zeros(length(input) + length(fir) - 1)

    s_in = lastindex(input)

    for n=eachindex(fir)
        sconv[n:s_in+n-1] .+= fir[n] .* input
    end

    sconv[eachindex(Si_mem)] .+= Si_mem

    return sconv
end
```

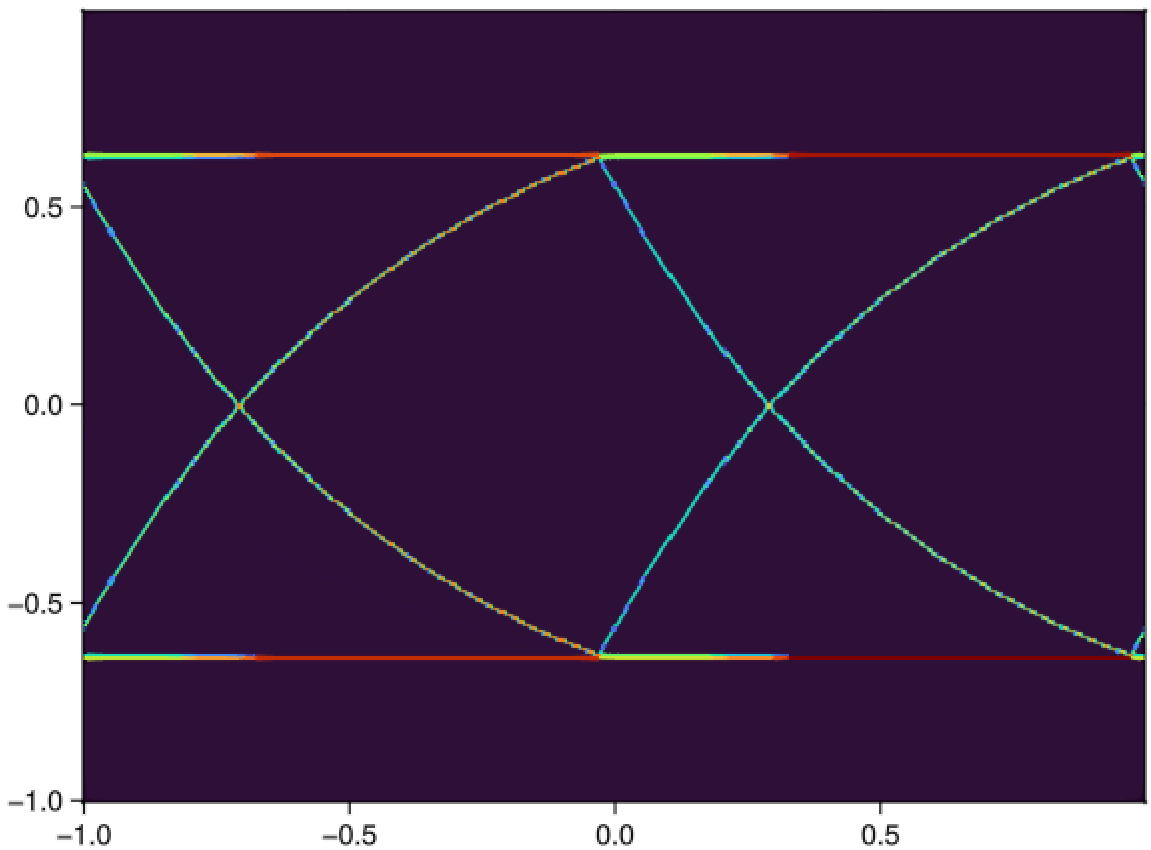
Now let's apply some FIR to open up the eye. Here we will use the non-mutating functions to maintain the states of the drv struct above.

```
s_filt = u_filt(bist.So, [1, -0.2]);
#change the FIR filter here to see the eye change reactively!
```

```
Vosr_filt = kron(s_filt, ones(param.osr));
```

```
Vo_filt_conv = u_conv(Vosr_filt, drv.ir, gain = drv.swing * param.dt);
```

```
eye_tx_filt = w_gen_eye_simple(Vo_filt_conv[1:param.blk_size_osr],
                                x_npts_ui, x_npts,
                                y_range, y_npts,
                                osr= param.osr, x_ofst=0);
```

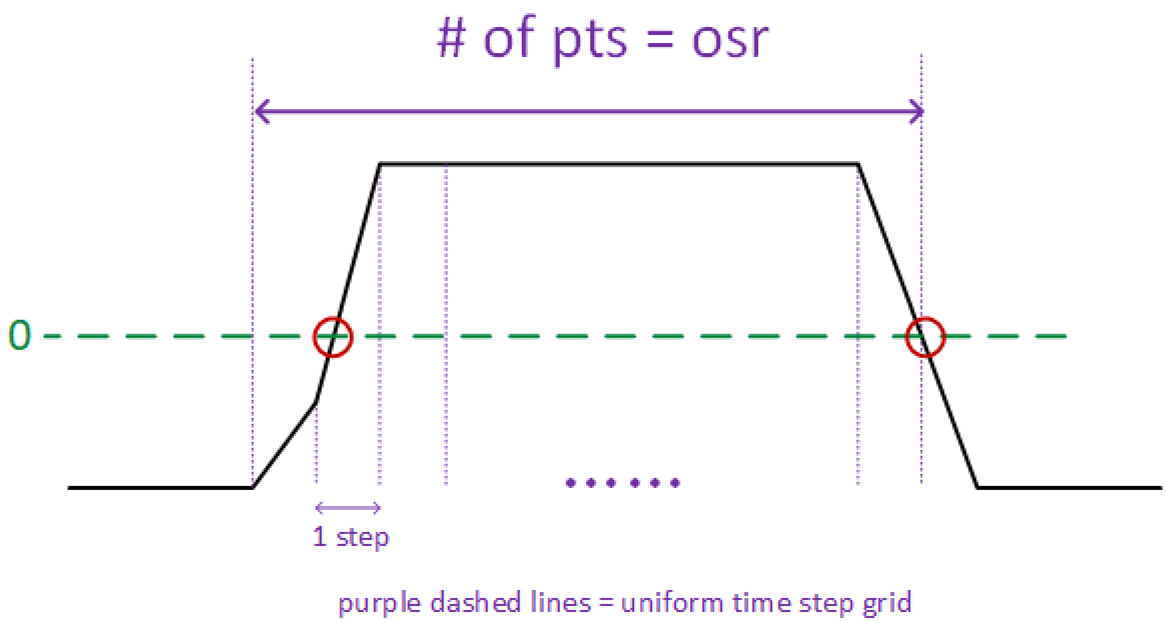


Cool! Actually, that was the easy part. Because we have defined a good `drv` struct and had clever uses of `views`, the functions seem quite straightforward and gave us interesting results to play with immediately.

Let's be jittery

Too much coffee when I wrote this notebook? You bet! That's why our TX needs to be jittery now. For a refresher on the main types of TX jitter, click here for Prof. Palermo's lecture slides on jitter.

Jokes aside, modeling time domain phenomenon is always a challenge. How do we model jitter when our simulation time step is "fixed"? The key insight is that in our simulation framework, the zero crossing in the oversampled waveform is implicitly between the finite steps, i.e. @ t = N*osr+0.5. Despite our simulation time step being discrete, we can still use interpolation to "encode" where our zero crossing should be. Note that by using intermediate voltages, we can embed jitter information at fractional time steps.



The trick then is to **warp or remap** a "jittery time grid" onto our "uniform time grid", for lack of better terms. We will first generate the Δt at each nominal edge transition.

```
Δtt = zeros(param.blk_size); #clean slate no jitter for all symbols
```

Let's start with duty cycle distortion. When there is a distortion of `dcd`, it means all even edges shift by `dcd/2*osr`, and all odd edges shift by `-dcd/2*osr` (the sign here doens't really matter). It's also best to set the `blk_size` as an even number because we are modeling duty cycle.

```
Δtt[1:2:end] .+= drv.dcd/2*param.osr;
```

```
Δtt[2:2:end] .+= -drv.dcd/2*param.osr;
```

Now we add random jitter

```
rj_osr = drv.rj_s/param.tui * param.osr; #conver RJ unit from seconds to osr
```

```
Δtt .+= rj_osr .* randn(param.blk_size);
```

Time for sinusoidal jitter. We first define the phase of jitter (`phi_sj`), then pass it into a sine function.

```
sj_amp_osr = drv.sj_amp_ui * param.osr;
```

```
sj_freq_norm = drv.sj_freq * param.tui; #normalize SJ frequency to symbol rate
```

```
begin
phi_sj = (drv.last_sj_phi .+ (2π*sj_freq_norm) * (1:param.blk_size)) .% (2π);
drv.last_sj_phi = phi_sj[end]; #store the last phase for next block
end;
```

```
Δtt .+= sj_amp_osr .* sin.(phi_sj);
```

Julia Tips

Julia supports syntax like `2π`! It will understand it as `2*π`. Makes mathematical programming much cleaner. Latex like syntax (e.g. Φ_{sj}, created by \Phi + Tab + _s + Tab + _j + Tab) is also supported.

Now we have generated all the jitter information for each transition edge. Next step is to create the finer grid to go with our voltage waveform. First, we need to extend our Δtt vector with some more samples from previous UIs to avoid overflow/underflow.

```
begin
drv.Δtt_ext[eachindex(drv.Δtt_prev_nui)] .= drv.Δtt_prev_nui
drv.Δtt_ext[lastindex(drv.Δtt_prev_nui)+1:end] .= Δtt
end;
```

Note this is a similar style as our convolution with memory. The Δtt_prev_nui vector is automatically taken care of for the next block due to views. We will also define our `Vext` here.

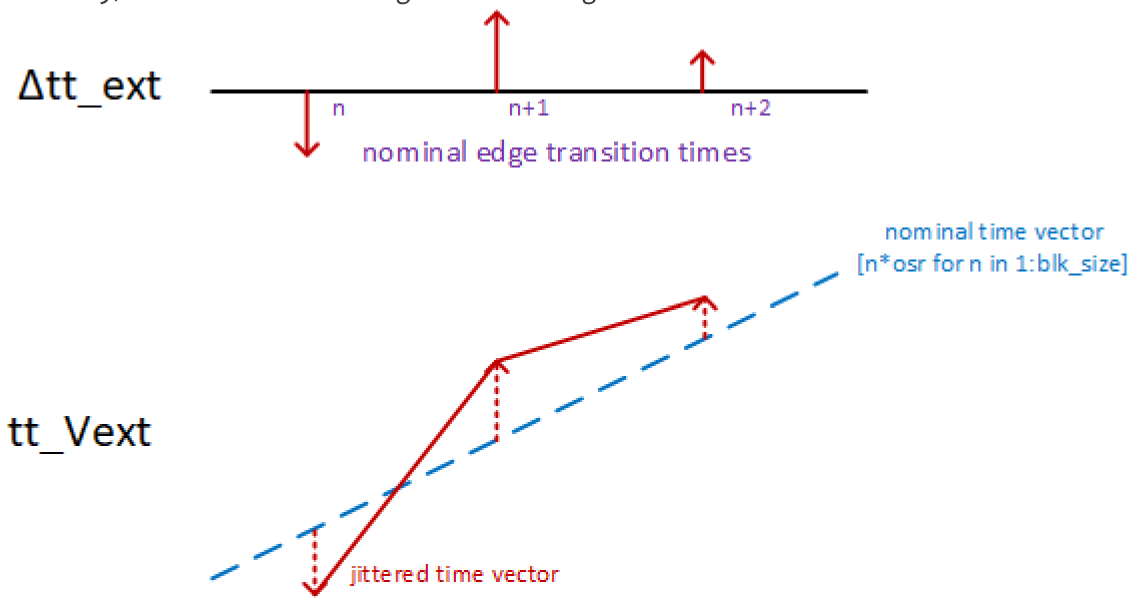
```
begin
drv.Vext[eachindex(drv.V_prev_nui)] .= drv.V_prev_nui
drv.Vext[lastindex(drv.V_prev_nui)+1:end] .= Vosr
    #we will just reuse the bit sequence above
end;
```

We now build a helper function to create a linear grid in between the transition times.

```
drv_jitter_tvec! (generic function with 1 method)
```

```
function drv_jitter_tvec!(tt_Vext, Δtt_ext, osr)
    for n = 1:lastindex(Δtt_ext)-1
        tt_Vext[(n-1)*osr+1:n*osr] .= LinRange((n-1)*osr+Δtt_ext[n],
n*osr+Δtt_ext[n+1], osr+1)[1:end-1]
    end

    return nothing
end
```

Pictorially, this function is doing the following




```
drv_jitter_tvec!(drv.tt_Vext, drv.Δtt_ext, param.osr);
```

Great! Now we created a jittered time axis for our voltage waveform. If we plot `tt_Vext` vs. `Vext`, this will represent the waveform with jitter! That's because we are viewing this plot from a uniform time grid perspective. So the last step then is to remap this signal back onto our uniform simulation time grid, through (drum roll....) **interpolation**.

Julia has a interpolation package, and slightly different way of doing interpolation compared to MATLAB. We will stick with the simple linear interpolation since accuracy can always be adjusted with increasing `osr`.

```
itp = linear_interpolation(drv.tt_Vext, drv.Vext); #itp is a function object
```

Julia's interpolation return a *function object* that can operate on any values you throw at it. Compared to MATLAB, the `interp` function takes the old axis/value and new axis together. Julia's interp function object comes handy when repeated interpolation is needed.

```
tt_uniform = (0:param.blk_size_osr-1) .+ drv.prev_nui/2*param.osr;
#note here tt_uniform is shifted by prev_nui/2 to give wiggle room for sampling "before" and "after" the current block. This is necessary for sinusoidal jitter
```

```
Vosr_jittered = itp.(tt_uniform);
#To interpolate, use the itp object like a function and broadcast to a vector
```

Julia Tips

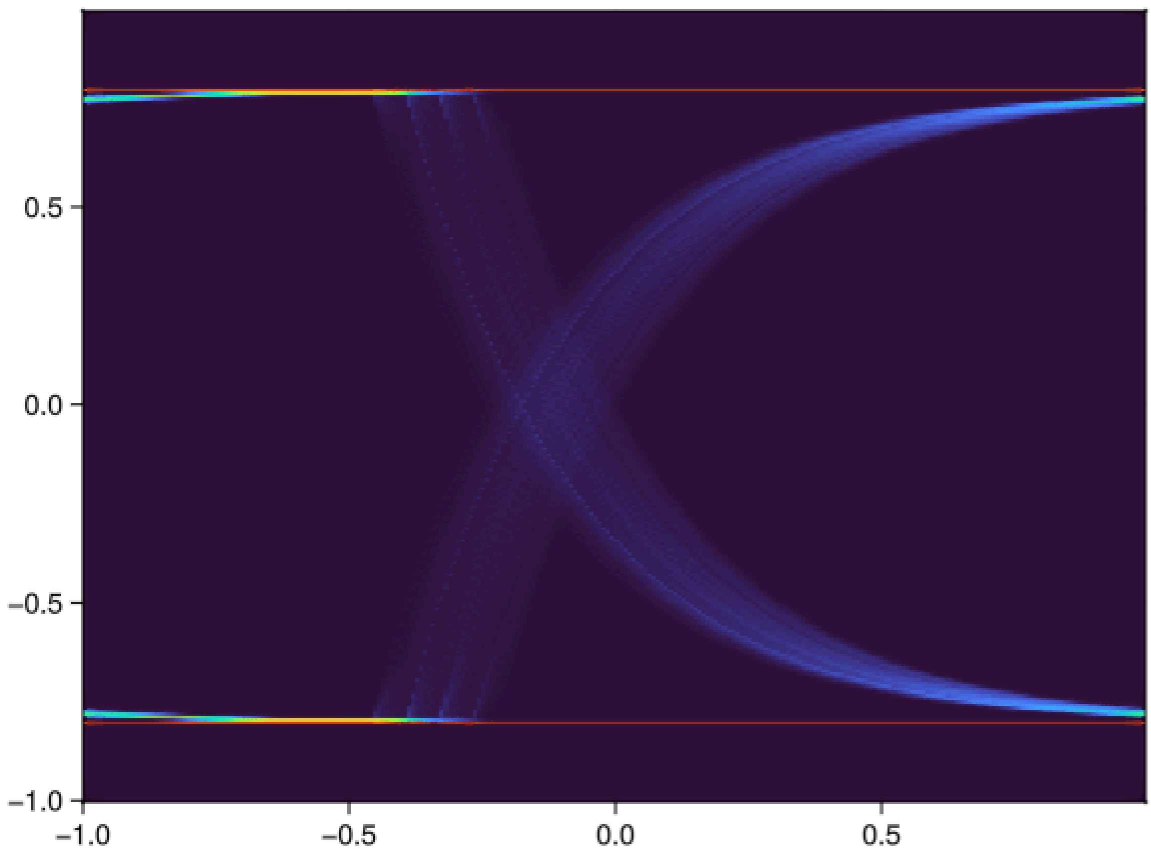
Julia can be C/C++ like if you want to optimize for performance. Using the built-in `linear_interpolation` can make your code functional at first, but might have too much overhead (i.e. checking for conditions and inputs to make the right internal call). It's possible for you to write a specialized interpolation function for this case. As an example, check out the `drv_interp_jitter!` function in the appendix.

To summarize and not interfere with previous codes in the notebook, you can play around with the code cell below to see how the jittered eye diagram change.

```
begin
#parameters for you to play around with
dcd = 0.05;
rj_s = 1000e-15;
sj_amp_ui = 0.02;
sj_freq = 10e6;

#unit conversion
rj_osr1 = rj_s/param.tui*param.osr
sj_amp_osr1 = sj_amp_ui*param.osr
sj_freq_norm1 = sj_freq*param.tui

Δtt1 = zeros(param.blk_size);
Δtt1[1:2:end] .+= dcd/2*param.osr;
Δtt1[2:2:end] .-= dcd/2*param.osr;  #add dcd
Δtt1 .+= rj_osr1 .* randn(param.blk_size); #add rj
phi_sj1 = (0.0 .+ (2π*sj_freq_norm1) * (1:param.blk_size)) .% (2π);
Δtt1 .+= sj_amp_osr1 .* sin.(phi_sj1); #add sj

#gen jittered time
drv.Δtt_ext[eachindex(drv.Δtt_prev_nui)] .= drv.Δtt_prev_nui
drv.Δtt_ext[lastindex(drv.Δtt_prev_nui)+1:end] .= Δtt1
drv_jitter_tvec!(drv.tt_Vext, drv.Δtt_ext, param.osr);
#voltage
drv.Vext[eachindex(drv.V_prev_nui)] .= drv.V_prev_nui
drv.Vext[lastindex(drv.V_prev_nui)+1:end] .= Vosr

#interpolate and remap to simulation grid.
#compare and see the speed difference between the custom interpolation and the built-in one when you know the problem
Vosr_jit1 = Vector{Float64}(undef, length(tt_uniform))
@time begin
itp1 = linear_interpolation(drv.tt_Vext, drv.Vext); #itp is a function object
Vosr_jit1 = itp1.(tt_uniform);
end
@time drv_interp_jitter!(Vosr_jit1, drv.tt_Vext, drv.Vext, tt_uniform)

#convolve with ir
ir_high_bw = u_gen_ir_rc(param.dt, param.fbaud, 20*param.tui);
Vo_jit1 = u_conv(Vosr_jit1, ir_high_bw, gain = drv.swing * param.dt);
Vo_jit1_trunc = @views Vo_jit1[100*param.osr:end-100*param.osr]; #trim off some garbage for now
end;
```

```
0.108339 seconds (28 allocations: 17.049 MiB)
0.006312 seconds
```

```
eye_tx_jit = w_gen_eye_simple(Vo_jit1_trunc,
                            x_npts_ui, x_npts_ui,
                            y_range, y_npts,
                            osr= param.osr, x_ofst=round(x_npts_ui/3));
#To see duty cycle effect better, here the eye diagram is plotted over just 1UI
```

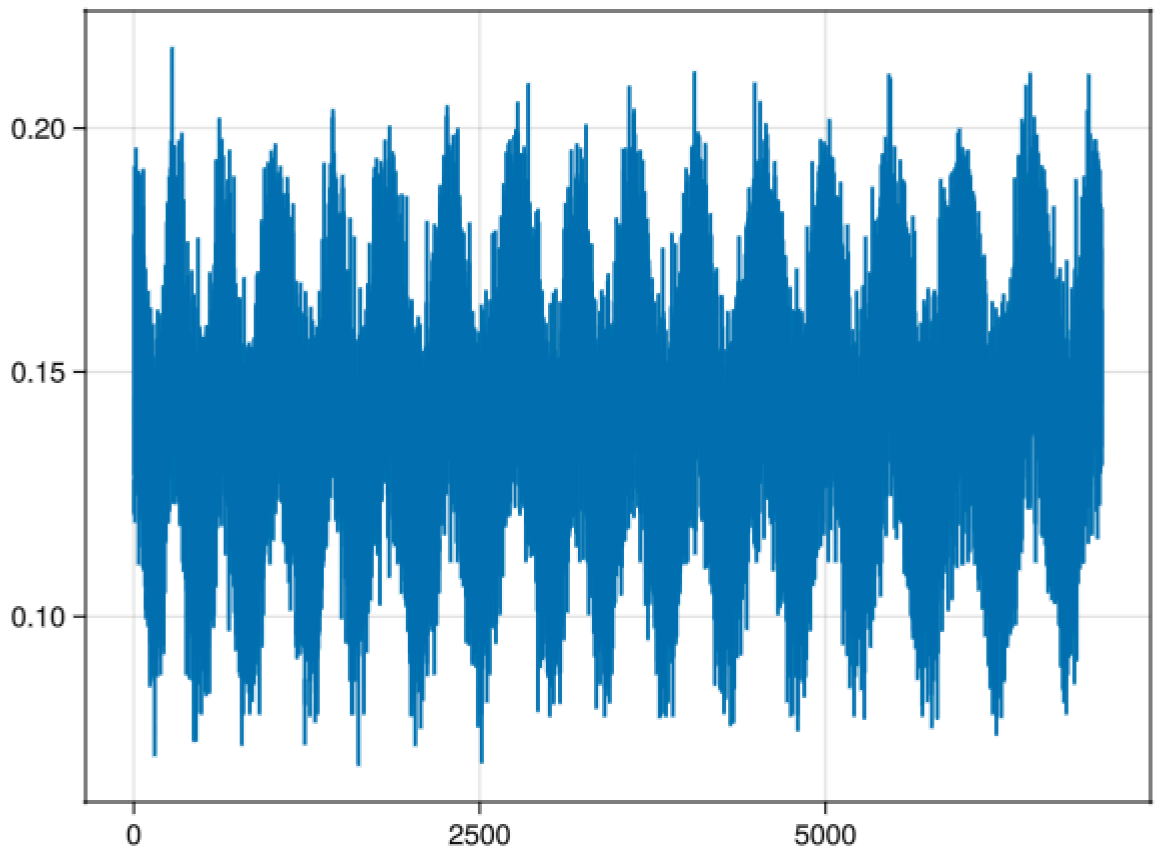


It's important to note that the impulse response of the driver (and subsequent channel, RX front end, etc.) plays a crucial role in low-pass filtering the jittered waveform to give it a "smoother look".

Analyzing jitter

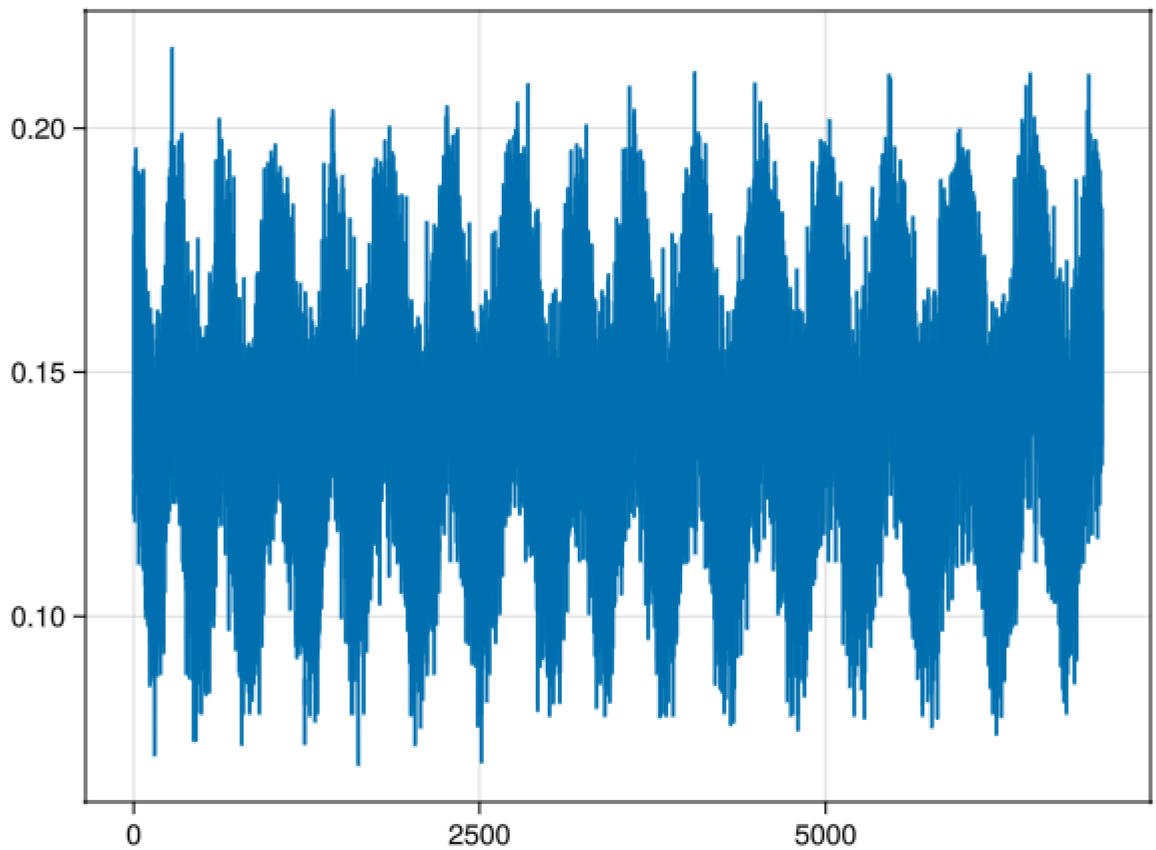
The utility module includes some helper functions to analyze jitter. We can use the find_0x (find zero crossing) function to generate jitter statistics of any waveform.

```
jit_0x = mod.(u_find_0x(Vo_jit1_trunc), param.osr) ./ param.osr;
#find zero crossing and normalize to within 1UI
```

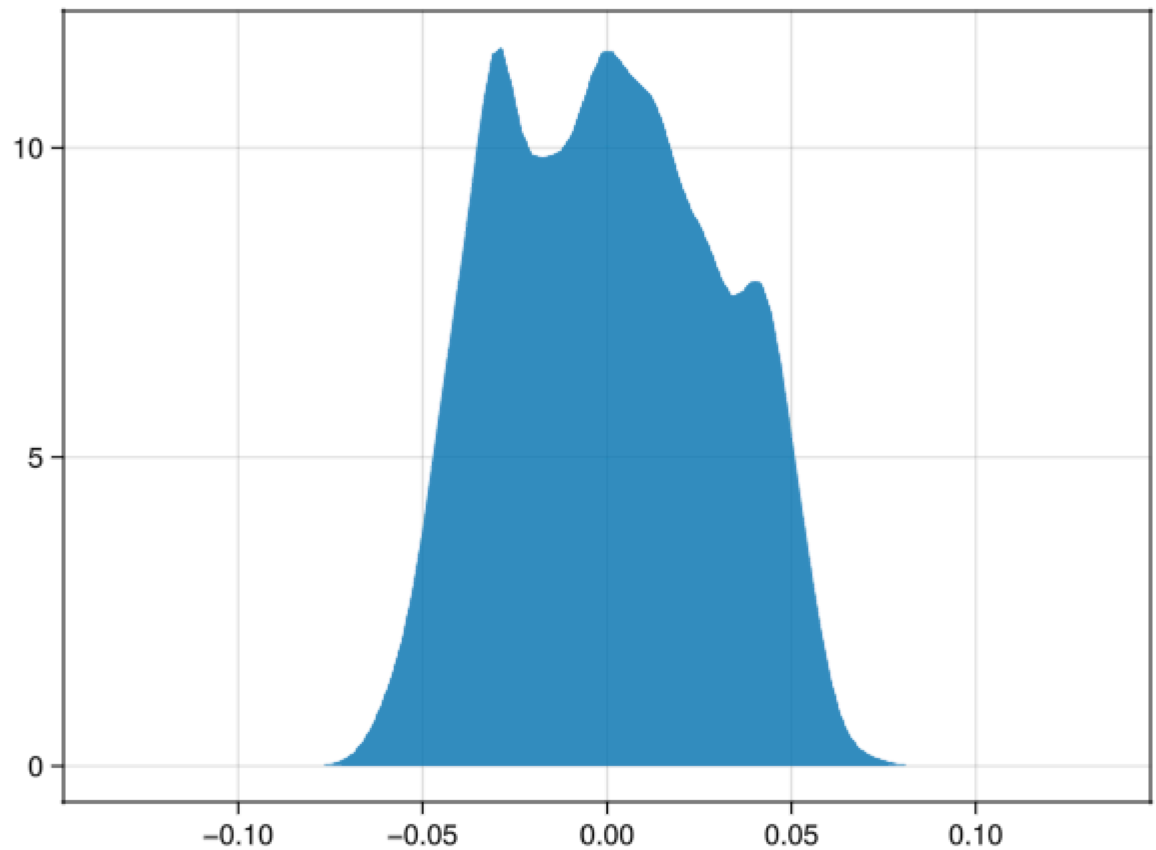


Because of normalization, there could be big jumps in the zero crossing points (try increasing the SJ amplitude and see). The unwrap_0x function in utility module can help unwrap these jumps.

```
jit_0x_unwrap = u_unwrap_0x(jit_0x);
```



Lastly, we can now plot the distribution of the jitter! Ideally, there could be another family of functions that take an distribution and extract dcd, RJ, SJ, etc. (just like a test instrument). All these functionalities could be added in the future (by anyone!).



Putting it all together

We now reach the point to put everything together. Below is the `dac_drv_top!` function showing everything we have covered. Refer to the source code in repository for more details on the `drv_add_jitter!` (which is just an encapsulation of the jitter section)

```
dac_drv_top! (generic function with 1 method)
```

```julia
function dac_drv_top!(drv, Si)
    @unpack osr, dt, blk_size, blk_size_osr = drv.param
    @unpack ir, fir_norm, swing, Sfir_mem, Vo_mem = drv

    #FIR filter
    u_filt!(drv.Sfir_conv, Si, fir_norm, Si_mem = Sfir_mem)

    #Oversample
    kron!(drv.Vfir, drv.Sfir, ones(osr))

    #Add jitter
    if drv.jitter_en
        drv_add_jitter!(drv, drv.Vfir)
    end

    #Convolve w/ ir
    u_conv!(drv.Vo_conv, drv.Vfir, ir, Vi_mem=Vo_mem, gain=dt*swing/2)

end
```

Conclusions

In this notebook, we covered the important concepts in a transmitter of model, including de-emphasis FIR, impulse responses, and jitter. A lot more is yet to be done, like nonlinearity and quantization.

The key modeling methods also directly apply to other circuit modules as well. Take advantage of views in Julia for more efficient use of memory. Convolution is a common function that will be shared among pretty much all blocks. Interpolation is important for transforming time axis as well as sampling (on the RX side).

Though not directly related to the simulation framework itself, plots are important tools when it comes to visualization and debug. We will cover more in depth about the plotting packages in Julia in the next one so that you can start generating great plots too!

Helper Functions

u_find_0x (generic function with 1 method)

```
function u_find_0x(input) #vectorized implementation
    sign_input = sign.(input)
    diff_sign = @views sign_input[2:end] .- sign_input[1:end-1]
    x_idx_crs = findall(abs.(diff_sign) .> 1 )
    x_idx_fine = Vector{Float64}(undef, lastindex(x_idx_crs))

    @. x_idx_fine = x_idx_crs+input[x_idx_crs]/(input[x_idx_crs]-input[x_idx_crs+1])

    return x_idx_fine
end
```

```
function u_find_0x(input) #explicit for-loop implementation
    sign_input = sign.(input)
    diff_sign = @views sign_input[2:end] .- sign_input[1:end-1]
    x_idx_crs = findall(abs.(diff_sign) .> 1 )
    x_idx_fine = Vector{Float64}(undef, lastindex(x_idx_crs))

    for n = eachindex(x_idx_crs)
        x_crs = x_idx_crs[n]
        x_idx_fine[n] = x_crs+input[x_crs]/(input[x_crs]-input[x_crs+1])
    end
    return x_idx_fine
end
```

u_unwrap_0x (generic function with 1 method)

```
function u_unwrap_0x(xpts; tol_Δui = 0.5) #assumes 0-1UI range, vectorized
    nwrap = 0
    xpts_unwrap = zeros(lastindex(xpts))
    xpts_unwrap[1] = xpts[1]

    ΔΦ = @views xpts[1:end-1] .- xpts[2:end]

    nwrap = cumsum( (abs.(ΔΦ) .> tol_Δui) .* sign.(ΔΦ))

    xpts_unwrap[2:end] .= nwrap .+ @views xpts[2:end]

    return xpts_unwrap
end
```

```julia
function u_unwrap_0x(xpts; tol_Δui = 0.5) #explicit for-loop implementation
    nwrap = 0
    xpts_unwrap = similar(xpts)
    xpt_prev = xpts[1]

    for n = eachindex(xpts)
        xpt = xpts[n]
        if abs(xpt-xpt_prev) > tol_Δui
            nwrap -= sign(xpt-xpt_prev)
        end

        xpts_unwrap[n] = nwrap + xpt

        xpt_prev = xpt

    end
    return xpts_unwrap
end
```

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

```julia
include("../../src/util/Util_JLSD.jl");
```

```julia
import .Util_JLSD: u_gen_ir_rc
```

```julia
include("../../src/blks/BlkBIST.jl");
```

```julia
import .BlkBIST: pam_gen_top!
```

```julia
include("../../src/blks/WvfmGen.jl");
```

```julia
import .WvfmGen: w_gen_eye_simple
```

```julia
using StatsBase, DSP, Interpolations, FFTW, MAT
```

```julia
using Parameters, DataStructures
```

```julia
using UnPack, Random, Distributions, BenchmarkTools
```

```julia
using GLMakie, Makie
```

Appendix

```
@kwdef mutable struct Param
    const data_rate::Float64
    const osr::Int64
    const pam::Int8 = 2
    const bits_per_sym::Int8 = Int(log2(pam))
    const fbaud = data_rate/bits_per_sym
    const fnyq= fbaud/2
    const tui = 1/fbaud
    const dt= tui/osr

    const blk_size::Int64
    const blk_size_osr::Int64 = blk_size*osr
    const subblk_size::Int64 = blk_size
    const nsubblk::Int64 = Int(blk_size/subblk_size)
    const nsym_total::Int64
    const nblk = Int(round(nsym_total/blk_size))

    const rand_seed = 300

    cur_blk = 0
    cur_subblk = 0

end



@kwdef mutable struct Bist
    const param::Param
    const polynomial::Vector{UInt8}
    const order::UInt8 = maximum(polynomial)
    const inv = false

    gen_seed = ones(Bool,order)
    gen_gray_map::Vector{UInt8} = []
    gen_en_precode = false
    gen_precode_prev_sym = 0

    chk_start_blk = 100
    chk_seed = zeros(Bool,order)
    chk_precode_prev_sym = 0
    chk_lock_status = false
    chk_lock_cnt = 0
    chk_lock_cnt_threshold = 128
    ber_err_cnt = 0
    ber_bit_cnt = 0

    So_bits::Vector = zeros(Bool, param.bits_per_sym*param.blk_size)
    So::Vector = zeros(param.blk_size)
    Si = CircularBuffer{UInt8}(param.blk_size)
    Si_bits::Vector = zeros(Bool, param.bits_per_sym*param.blk_size)
end
```

```
function u_gen_ir_rc(dt,bw,t_len)
    tt = [0:dt:t_len-dt;]

    ω = (2*π*bw)
    ir = ω*exp.(-tt*ω)
    ir = ir/sum(ir*dt)

    return ir
end
```

drv_interp_jitter! (generic function with 1 method)

```
function drv_interp_jitter!(vo, tt_jitter, vi, tt_uniform)
    last_idx = 1
    for n = eachindex(tt_uniform)
        t = tt_uniform[n]
        for m = last_idx:lastindex(tt_jitter)-1
            if (t >= tt_jitter[m]) && (t < tt_jitter[m+1])
                k = (vi[m+1]-vi[m])/(tt_jitter[m+1]-tt_jitter[m])
                vo[n] = vi[m] + k*(t-tt_jitter[m])
                last_idx = m
                break
            end
        end
    end

    return nothing
end
```