Basic DSP Modeling using Julia in Pluto Notebook

```
1 using PlutoUI
```

Table of Contents


```
1 TableOfContents(title="Table of Contents", indent=true, depth=5, aside=true)
```

From the signal processing perspective there are several computation and plots types that are of

interest. Like all computation languages, e.g., Python, Matlab/Octave, Mathematica, and even C/C++, need data types and functions for calculations and graphics. These need to be available from the notebook workspace. For the case of a Pluto notebook the Julia package `Pluto` needs to be installed and launched first.

For a text cell we 'hide' the underlying *markdown* code to just allow the formatted results to be seen. The output from a Notebook cell in Pluto is displayed above the code/markdown code.

Implement a Frequency Response Calculation

```
1 md"## Implement a Frequency Response Calculation"
```

Consider the following:

$$H(f) = \frac{1 + e^{-j2\pi f}}{2} = \frac{1}{2}\sum_{k=0}^{2-1} e^{-j2\pi k \cdot f},\ 0 \le f \le 0.5$$

where in DSP we typically use ω as the radians per sample frequency, but upon factoring out 2π we have the normalized frequency f in Hz/sample.

- Note Julia allows the use of greek letters in variables, e.g., `\pi` becomes π after hitting the tab key
- To obtain an imaginary number use `im`, so for $5j$ enter `5im`

```
π = 3.1415926535897...
```

```
1 π
```

```
[1.0-0.0im, 0.99999-0.00314157im, 0.999961-0.00628302im, 0.999911-0.00942422im, 0.999842
```

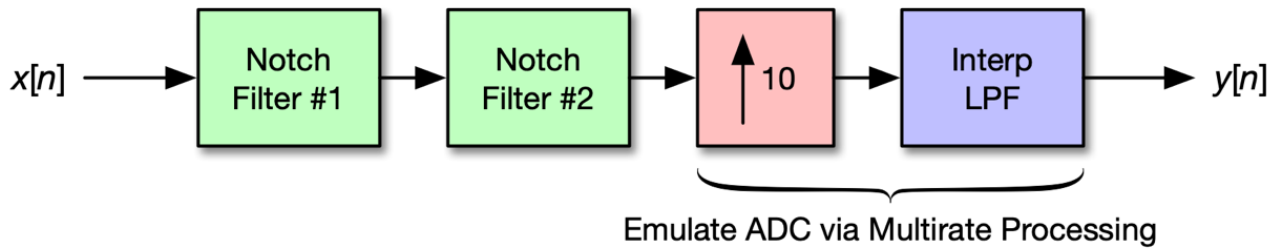
```
1 begin
2     f = collect(0:.001:0.5);
3     H = (1 .+ exp.(-im*2*pi*f))/2; # Note 1*im can be typed as 1im
4 end
```

Click the *disclosure triangle* in the above cell to display a more complete listing of the complex array `H`

Include a Figure

In Pluto a small function is created to display PNG figures. We then use it.

```
"support for image import"
```



System Block Diagram to Explore IIR Notch Filter Properties

```
Wow("iir_notch_block.png")
```

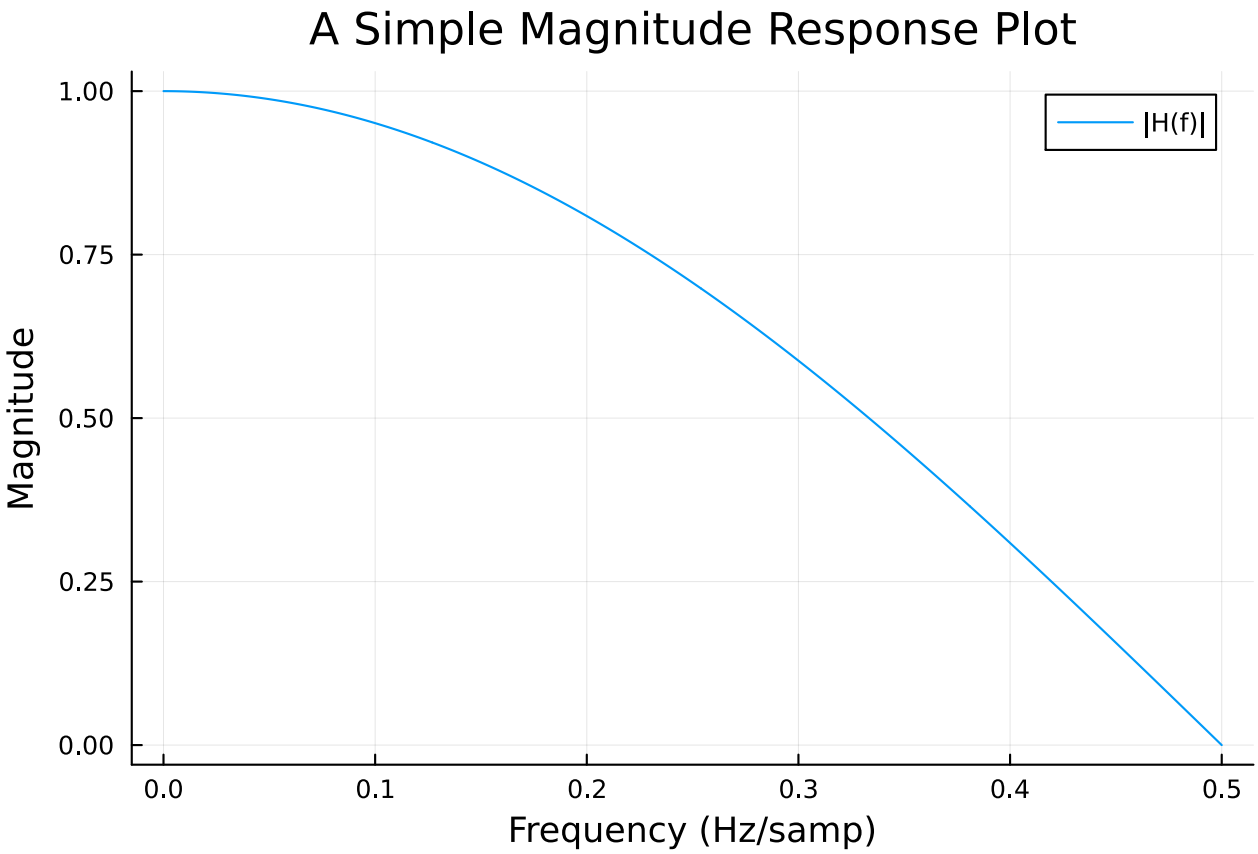
Plot $H(f)$

To create plots we need to import a plotting package. There are many such package available, but my preference is to use `Plots`. This package needs to added to the base install of Julia using the `package manager`. Start the Julia REPL (read-execute-print-loop) interface. Once inside the REPL type the key `]` to enter the package manager. The cursor will switch from `julia>` to `pkg>`. At the prompt enter `add Plots`. The `Plots`. The package will install from the cloud. When completed you can type `status` to see the packages installed. You can also type `up` to upgrade all packages and `remove Package_name` to remove a package. Note all package names begin with a Capital letter.

Finally load the package using either `using` or `import`. With `using` all of the names from the module/package gett loaded into the current workspace. With `import` you must type the full package name followed by dot following by the function name. In general `import` is better as it avoids name space pollution.

```
import Plots
```

- Note the import of `Plots` may throw a warning on Windows.




```
begin
    Plots.plot(f,abs.(H),label="|H(f)|",xlabel="Frequency
    (Hz/samp)",ylabel="Magnitude")
    Plots.plot!(title="A Simple Magnitude Response Plot") # use plots! to add more
    traces and/or more details
end
```

Generalize the Filter Order and Write a Function in Julia

We now consider an $N+1$-tap causal moving average filter and its associated frequency response

$$H(f) \overset{\text{also}}{=} H\left(e^{j2\pi f}\right) = \frac{1}{N+1}\sum_{k=0}^{N} e^{-j2\pi k \cdot f},\ 0 \le f \le 0.5$$

Note $N+1$ total terms in the sum. The calculation will be wrapped in a *function* that uses a loop structure to sum the finite series.

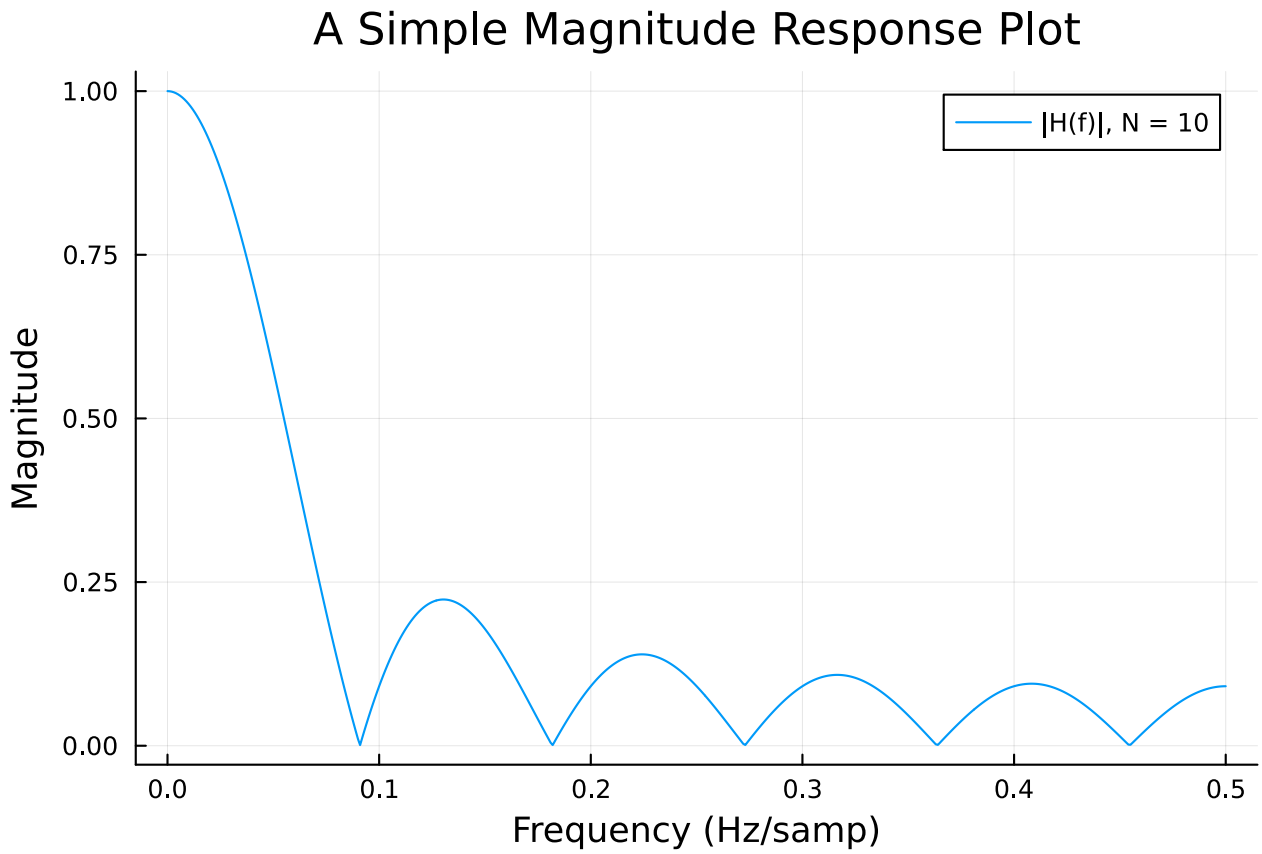
MA_freq_resp

```
H_N, f = MA_freq_resp(N)
```

A function that obtains the frequency response of a MA filter

```
"""
    H_N, f = MA_freq_resp(N)

A function that obtains the frequency response of a MA filter
"""
function MA_freq_resp(N)
    f = collect(0:.001:.5);
    H_N = zeros(length(f))
    for k in range(0,stop=N)
        # H_N = H_N + exp.(-1im*2*pi*k*f);
        H_N += exp.(-1im*2*pi*k*f);
        # println(k)
    end
    H_N /= (N+1)
    return H_N, f
end
```




```
begin
    N = 10
    H_N, f2 = MA_freq_resp(N) # We use f2 over f to avoid a duplicate variable

    Plots.plot(f2,abs.(H_N),label="|H(f)|, N = $N",xlabel="Frequency
    (Hz/samp)",ylabel="Magnitude")
    Plots.plot!(title="A Simple Magnitude Response Plot") # use plots! to add more
    traces and/or more details
end
```

To introduce another plot type, you can establish that the causal $N+1$ tap moving average filter has impulse response

$$h[n] = \frac{1}{N+1}\sum_{k=0}^{N}\delta(n-k) = \frac{1}{N+1}\big(u[n]-u[n-(N+1)]\big)$$

A *stem* plot is a good way to plot the impulse response. We can agin use `Plots` with line style options.

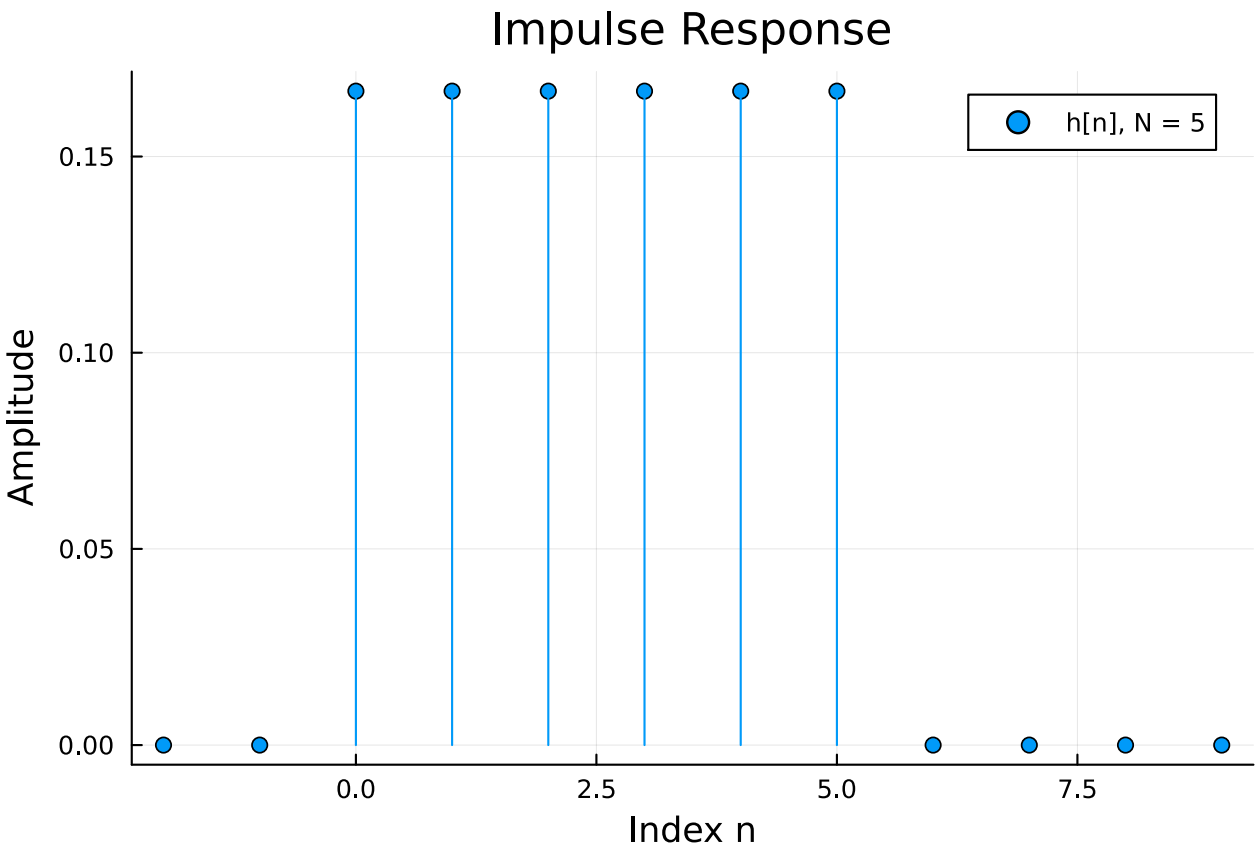
MA_imp

```
h, n = MA_imp(N; n_before=2,n_after=4)
```

A function that obtains the impulse response of a MA filter,

```
"""
    h, n = MA_imp(N; n_before=2,n_after=4)

A function that obtains the impulse response of a MA filter,
"""
function MA_imp(N; n_before=2,n_after=4)
    n = collect(-n_before:N+n_after)
    h = vcat(zeros(n_before),ones(N+1)/(N+1),zeros(n_after))
    return h, n
end
```



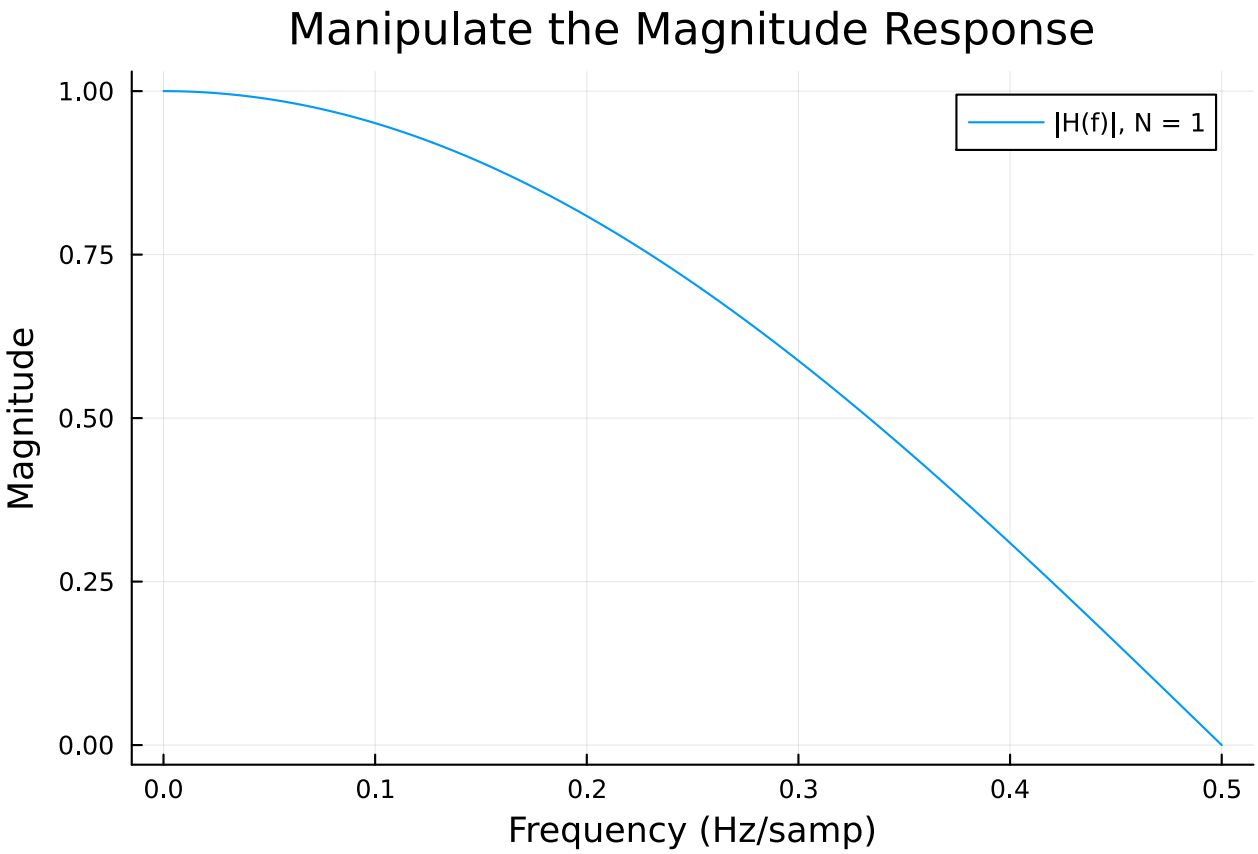

```
begin
    N2 = 5
    h, n2 = MA_imp(N2)
    Plots.plot(n2, h, label="h[n], N = $N2", line=:stem, marker=:circle,
    xlabel="Index n", ylabel="Amplitude", title="Impulse Response")
end
```

Interactive Frequency Response using the `PlutoUI` Widgets

To make an analysis model interactive in `Pluto` notebooks require just the installation of the package `PlutoUI`

Widgets such as sliders are created in markdown cells and are connected to variables using `@bind`. Following examples is the best way to make this work.zz

Change the MA filter order: N = 1 Taps

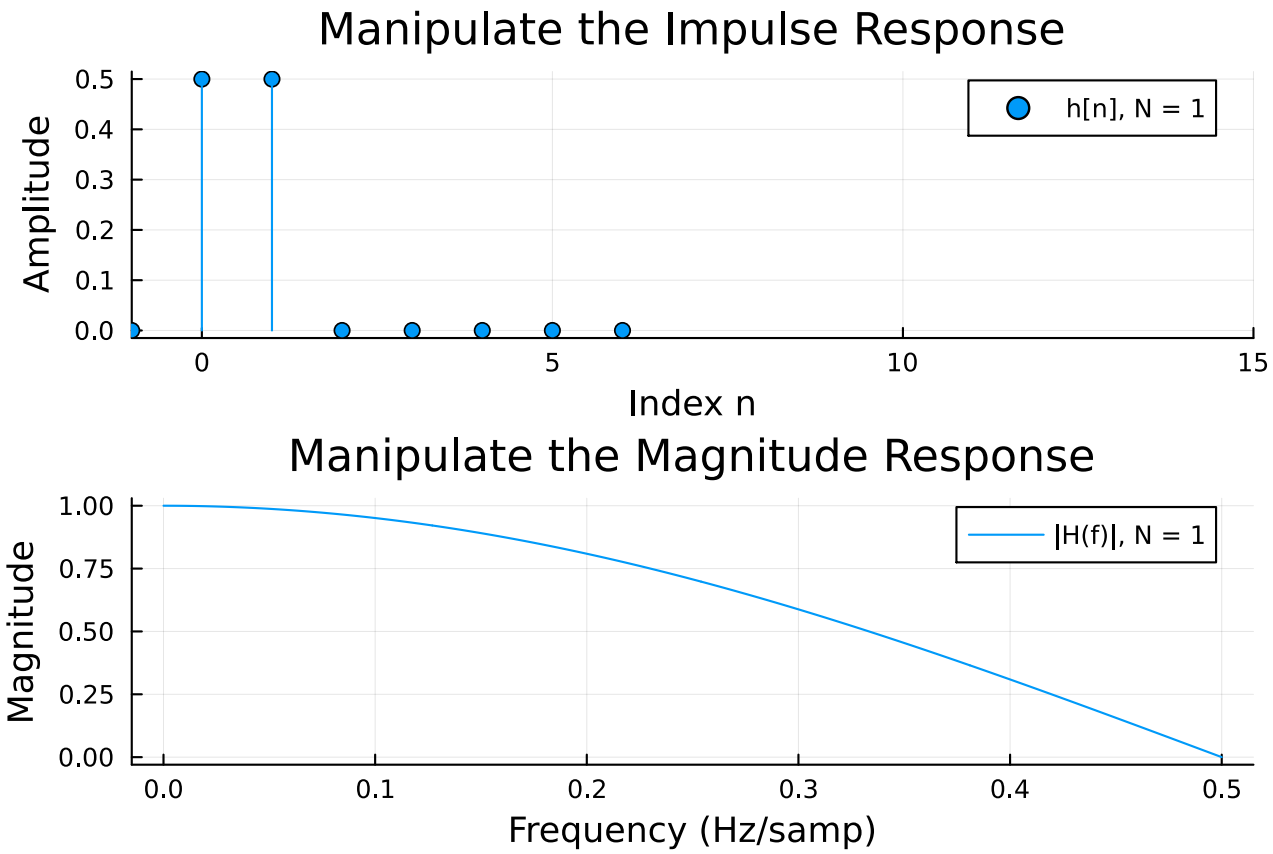



```
begin
    H_Na, fa = MA_freq_resp(N_a)
    Plots.plot(f,abs.(H_Na),label="|H(f)|, N = $N_a",xlabel="Frequency (Hz/samp)",ylabel="Magnitude")
    Plots.plot!(title="Manipulate the Magnitude Response") # use plots! to add more traces and/or more details
end
```

Interactive Frequency Response and Impulse Response using a Plots `layout`

We will extend the above `PLutoUI` slider example to contain a layout of two plots:

Change the MA filter order: N = 1 Taps




```
begin
    l = Plots.@layout [top; bot]; # define a plot layout for multiple plots
    h_Nal, n_Nal = MA_imp(N_b;n_before=1,n_after=5)

    H_Nal, f_Nal = MA_freq_resp(N_b)
    top = Plots.plot(n_Nal, h_Nal, label="h[n], N = $N_b", line=:stem,
marker=:circle, xlabel="Index n",
            ylabel="Amplitude", title="Manipulate the Impulse Response",xlim=(-1,15))
    bot = Plots.plot(f_Nal,abs.(H_Nal),label="|H(f)|, N = $N_b",xlabel="Frequency
(Hz/samp)",ylabel="Magnitude")
    bot = Plots.plot!(title="Manipulate the Magnitude Response")
    Plots.plot(top, bot, layout = l); # display the plot layout
end
```

Use Custom Julia Code Modules for DSP and Comm

The Julia community has package authors developing packages for both signal processing and communications modeling. A better starting point however to the package listing WebSite: https://juliapackages.com/packages. The default `stargazers` listing option on the site lists the packages in a desending order of the *stars* the packages have received from users.

Of the current top 20 packages (as of July 2022) packages that help me in my work the most are: Pluto, IJulia, Plots, and PyCall.

ingredients

```
alias_package_name = ingredients(path::String)
```

This function is used to import a user code module into the current workspace.


```
alias_package_name = ingredients(path::String)
```

This function is used to import a user code module into the current workspace.


```
1 include("modules/ingredients_function.jl")
```

Import custom code modules developed to provide support functions for DSP and digital communications modeling in Julia. The modules `DSPTools.jl` and `CommTools` are under development by Mark Wickert. The functions in these modules rely upon the standard Julia packages and other packages as you can see from the `import` statements below:

```
import Plots
import Polynomials
import FFTW
import DSP
```

When using these modules in the Pluto notebook you have to explicity import the modules into the notebook, in spite of the fact that they are imported by the module itself.

In some cases the functions provide interfaces more like similar functions found in Python and Octave/Matlab. We now import the modules.

```
1 import Polynomials, FFTW, DSP
```

```
DSPTools = Main.var"DSPTools.jl"
```

```
1 DSPTools = ingredients("modules/DSPTools.jl")
```

```
CommTools = Main.var"CommTools.jl"
```

```
1 CommTools = ingredients("modules/CommTools.jl")
```

```
PLLTools = Main.var"PLLTools.jl"
```

```
1 PLLTools = ingredients("modules/PLLTools.jl")
```

To see the functions in these modules you can look at the modules right in the `Jupyter Lab` IDE and `vs code`. You can also see a listing right here in Jupyter by typing `DSPTools.` and pressing the tab key:

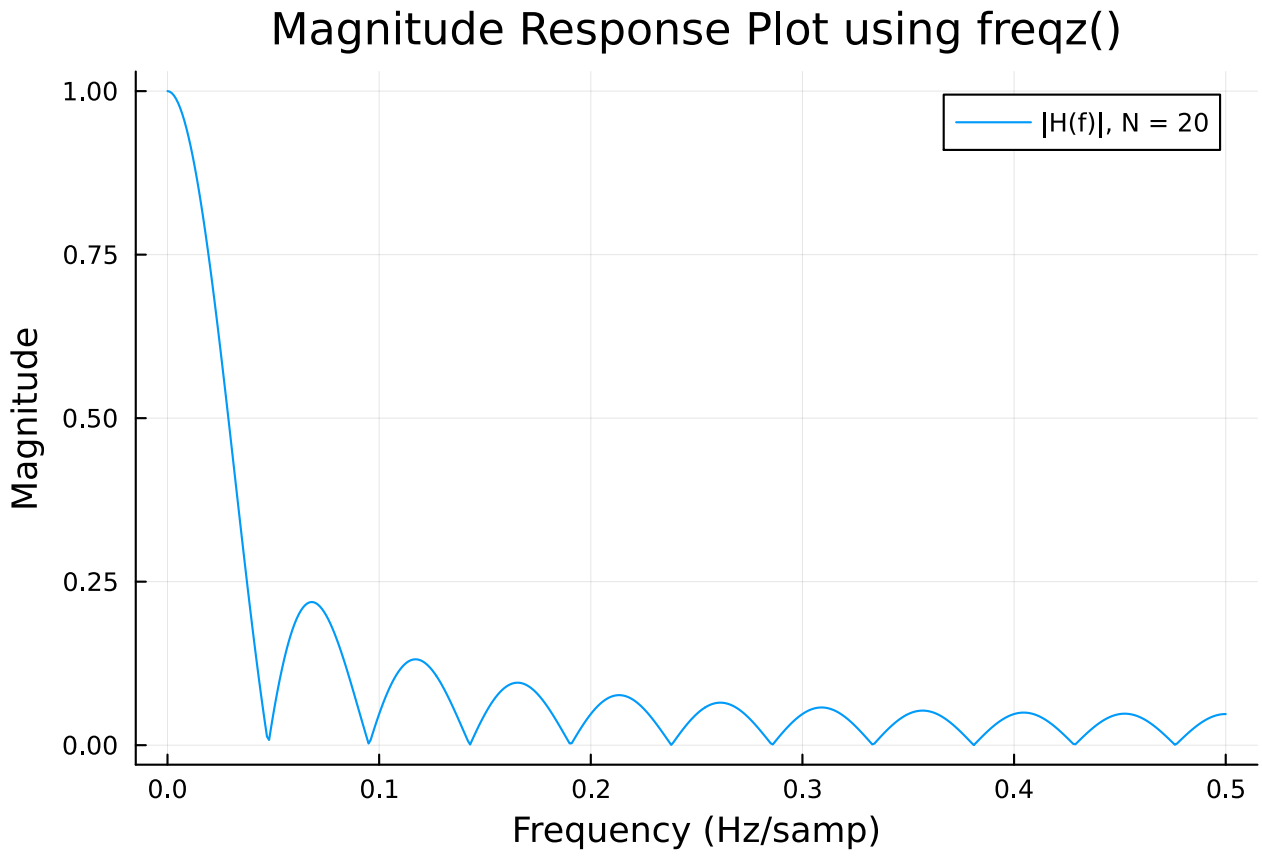
```
#DSPTools.
```

```
#CommTools.
```

Note: `CommTools` list more functions because it also imports the `DSPTools` modules.

To see docstring help for a given function place the cursor inside the function and open up Pluto's `Live docs` in the lower right of the browser window.

Use `DSPTool.freqz()` to Find the Frequency Reponse of a 20 Tap MA Filter




```
begin
    N3 = 20
    H_N_dsp = DSPTools.freqz(ones(N3+1)/(N3+1),[1],f)
    Plots.plot(f,abs.(H_N_dsp),label="|H(f)|, N = $N3",xlabel="Frequency
    (Hz/samp)",ylabel="Magnitude")
    Plots.plot!(title="Magnitude Response Plot using freqz()")
end
```

Importing and Using Python Packages via `PyCall`

To take advantage of the vast number of packages available for Python eco-system we can use `PyCall`. This will require the installation of the `PyCall` package and likely the `Conda` package. The `Conda` package installs a local version of Python via `miniconda`. Popular Python packages such as `numpy`, `scipy` and `scikit-dsp-comm` can be installed

I say *likely* because in setting up `IJulia` the Conda package for Julia is required to provide *Jupyter Lab* which sits on top of a Python install. If you already have ...

```
using PyCall
```

```
ss =
PyObject <module 'sk_dsp_comm.sigsys' from '/Users/mwickert/.julia/conda/3/x86_64/lib/pyt
```

```
1 ss = pyimport("sk_dsp_comm.sigsys")
```

```
signal =
PyObject <module 'scipy.signal' from '/Users/mwickert/.julia/conda/3/x86_64/lib/python3.1
```

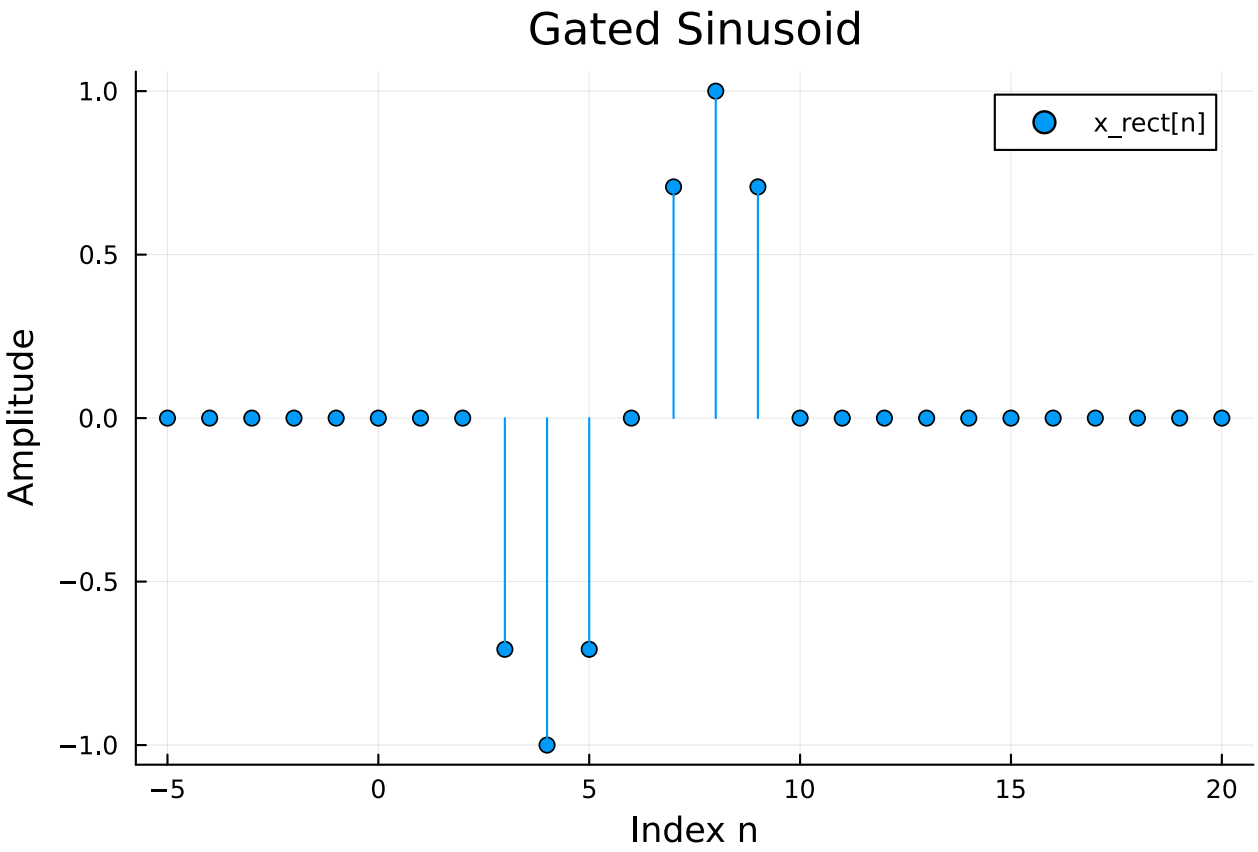
```
1 signal = pyimport("scipy.signal")
```

```
np =
PyObject <module 'numpy' from '/Users/mwickert/.julia/conda/3/x86_64/lib/python3.10/site-
```

```
1 np = pyimport("numpy")
```

Using Signal Primitives from `sk_dsp_comm.sigsys`

Create a pulse gated sinusoid using `ss.dstep`




```
begin
    n_rect = collect(-5:20)
    wind_width = 8
    wind_start = 2
    x_rect = cos.(2π*n_rect/8) .*
             (ss.step(n_rect .- wind_start) - ss.step(n_rect .-
    (wind_start+wind_width)))
    Plots.plot(n_rect, x_rect, label="x_rect[n]", line=:stem, marker=:circle,
    xlabel="Index n", ylabel="Amplitude", title="Gated Sinusoid")
end
```

Note: The cell below can be *enabled* or *disabled* to control the placement of page breaks in the rendered pdf output. When the code is hidden you never see the code in the PDF output.

Why: This useful for insuring break breaks occur occur at the start of new homework problems. Placing computer modeling results following handwritten solutions is a nice to backup your work.

```
html"""
<script>
let cell = currentScript.closest("pluto-cell");
cell.style.pageBreakAfter = "always";
</script>
"""
```

5th-Order Butterworth Design

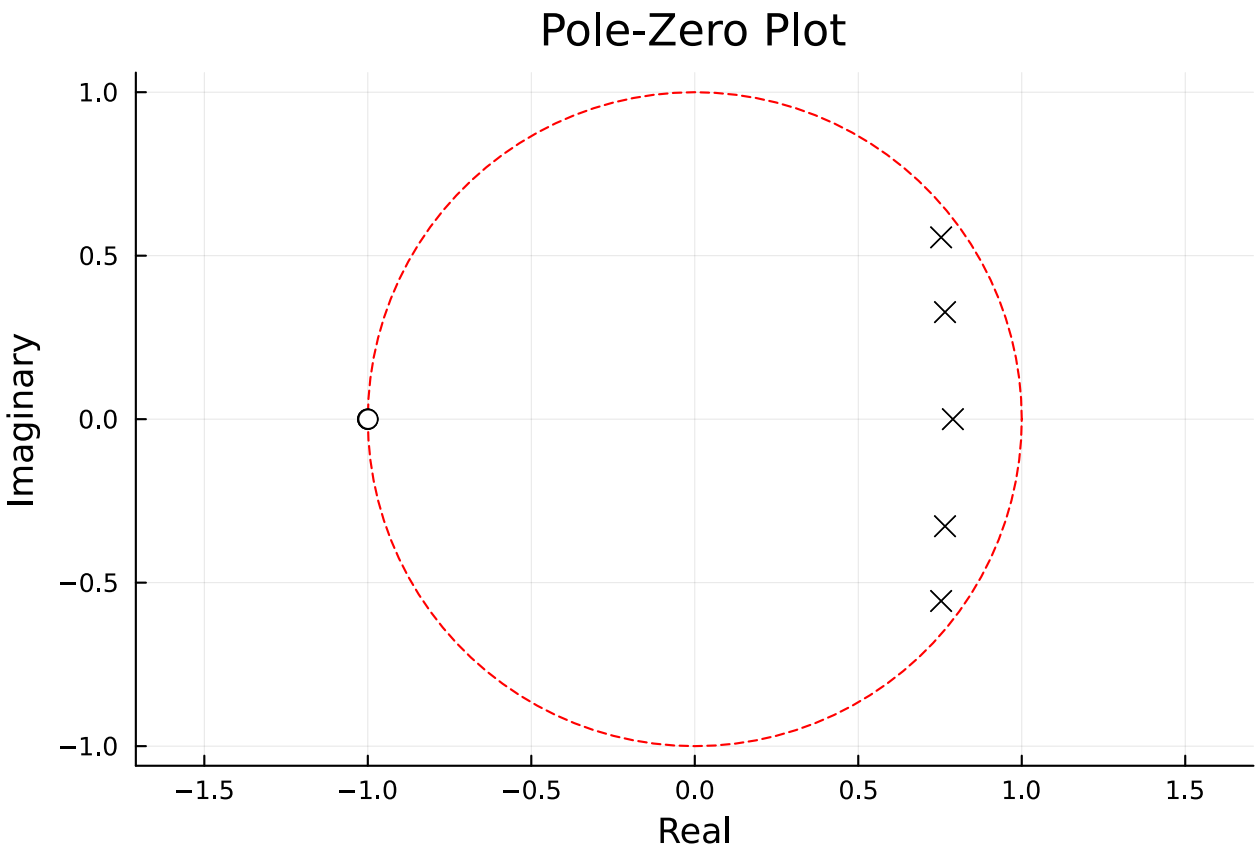
Begin by designing a Butterworth lowpass filter from it order N and its Nyquist rate normalized cutoff frequency $2 \times f_c/f_s$, where f_c is the cutoff frequenct in Hz and f_s is the sampling frequency in Hz. The linear constant coefficient difference equation (LCCDE) coefficients b, a arrays are returned using `scipy.signal.butter()`. The Julia `DSP` package could also be used but is less familiar to Python and MATLAB users.

Finally plot the pole-zero plot, impulse response, and frequency response. For this we use functions in `DSPTools`.

```
([0.000395228, 0.00197614, 0.00395228, 0.00395228, 0.00197614, 0.000395228], [1.0, -3.826
```

```
# b, a = signal.butter(5,2*0.1)
b, a = signal.cheby1(5,0.5,2*0.1)
```

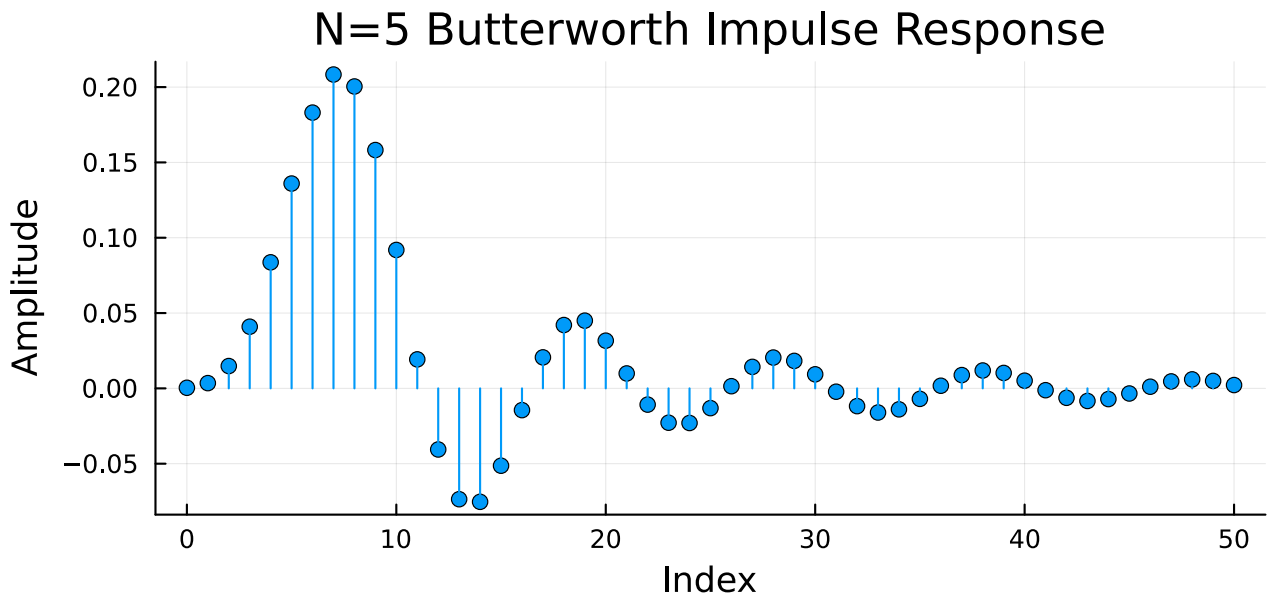
Pole-Zero Plot




```
DSPTools.zplane(b,a)
```

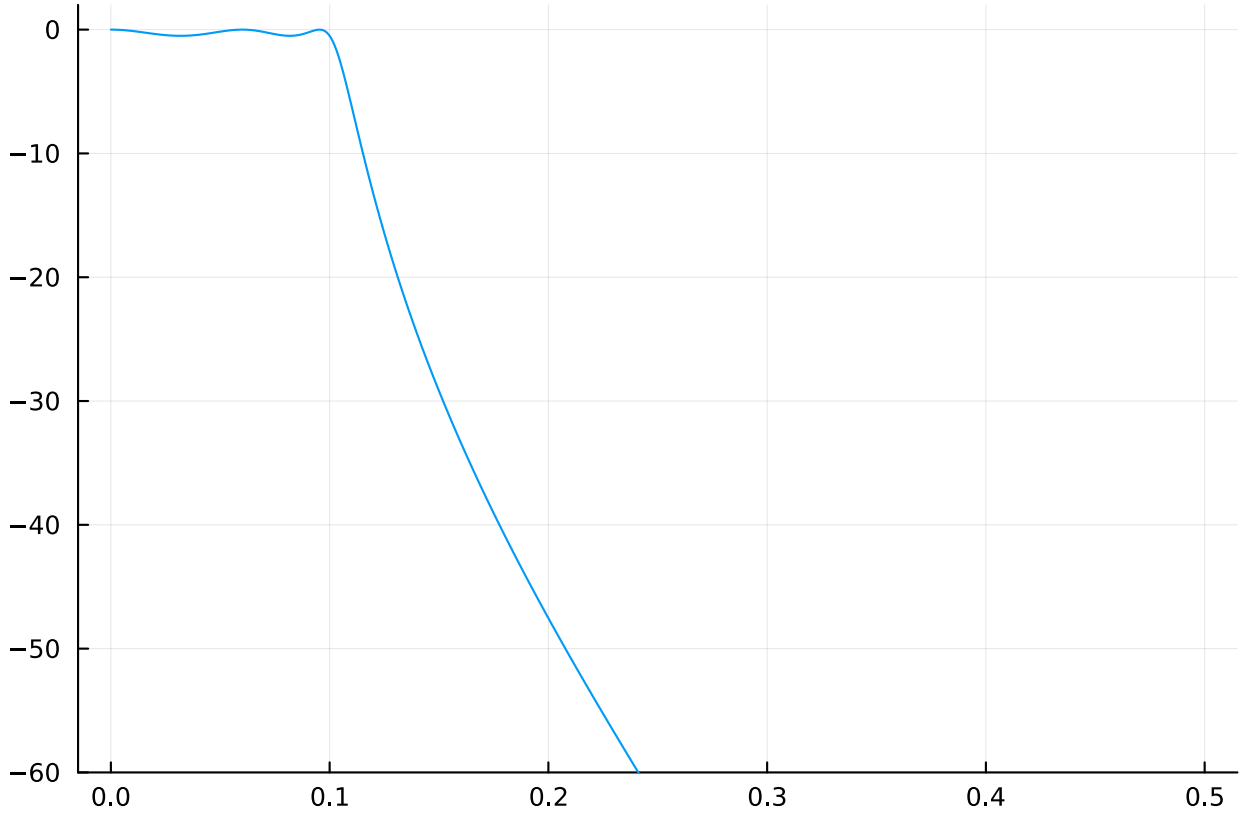
Impulse Response

- Drive the input with $\delta[n]$ and collect the output $h[n]$:




```
begin
    n_imp = collect(0:50)
    δ_seq = vcat([1], zeros(50)) # 1 followed by 50 zeros
    h_but_imp = DSPTools.lccde(b,a,δ_seq)
    DSPTools.stem(n_imp,h_but_imp;title="N=5 Butterworth Impulse Response")
end
```

Frequency Response



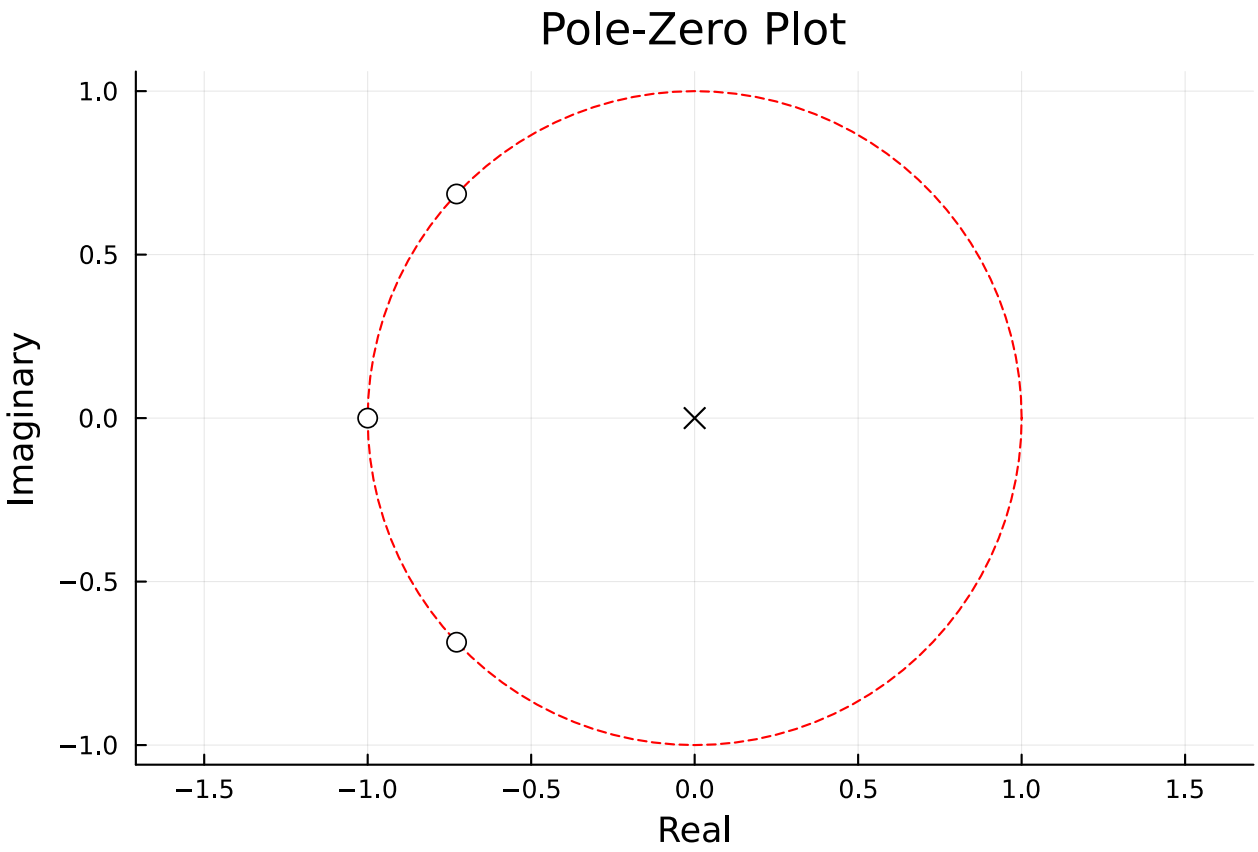

```
begin
    f_freqz = collect(0:1/1024:0.5)
    H_but = DSPTools.freqz(b,a,f_freqz)
    Plots.plot(f_freqz,20*log10.(abs.(H_but)),leg=false,ylim=(-60,2))
end
```

Three Zeros on the Unit Circle

Consider the system

$$H(z) = (1+z^{-1})(1-e^{j(3\pi/4+\theta_\epsilon)}z^{-1})(1-e^{j(5\pi/4-\theta_\epsilon)}z^{-1}),\ \theta_\epsilon \in \{-10°, 10°\}$$

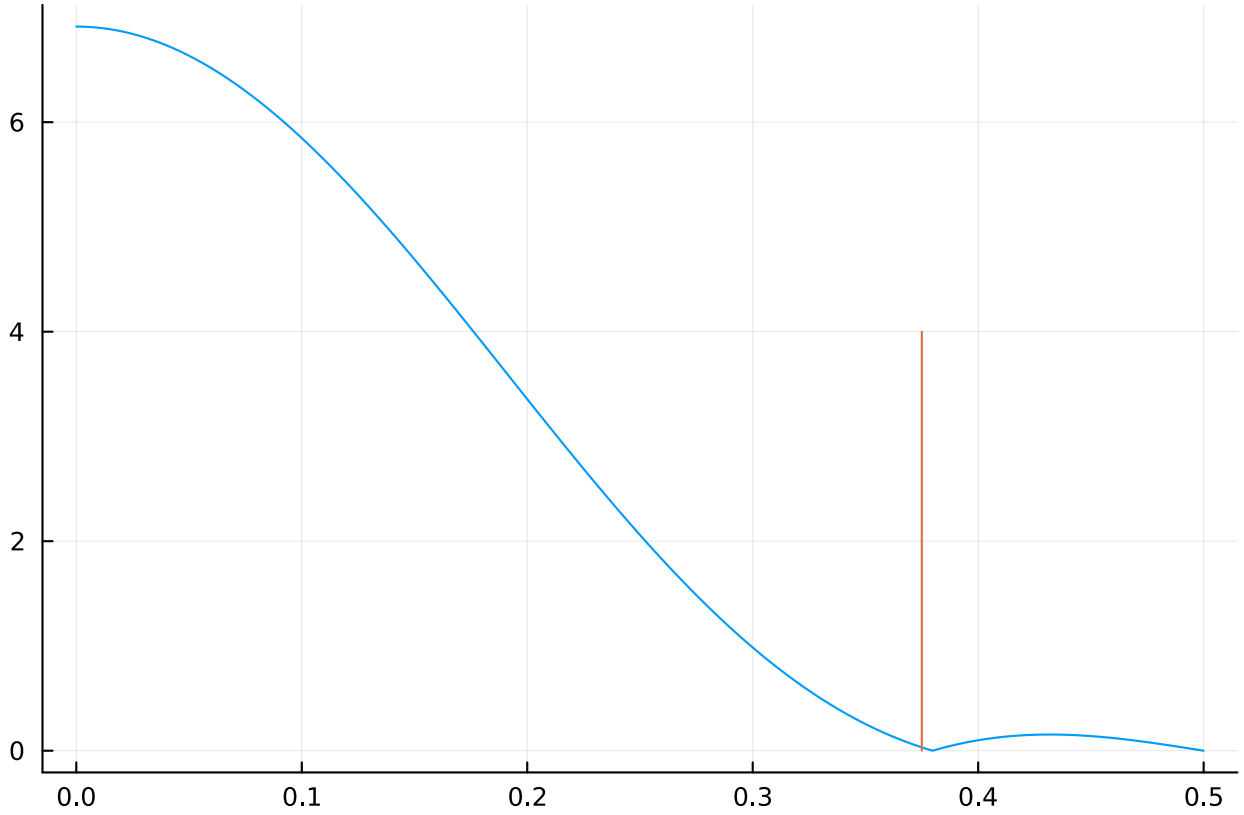
```
b_fnl = DSP.conv(DSP.conv([1,-exp(im*π)],[1,-exp(im*(3π/4+ D_angle*π/180))]),[1,-
exp(im*(5π/4-D_angle*π/180))]);
```




```
1 DSPTools.zplane(b_fnl,[1])
```

Fine Tuning Zeros on the Unit Circle

Change the angle about $\pi \pm \pi/2$: $\Delta\theta$ = 1.73 Degrees



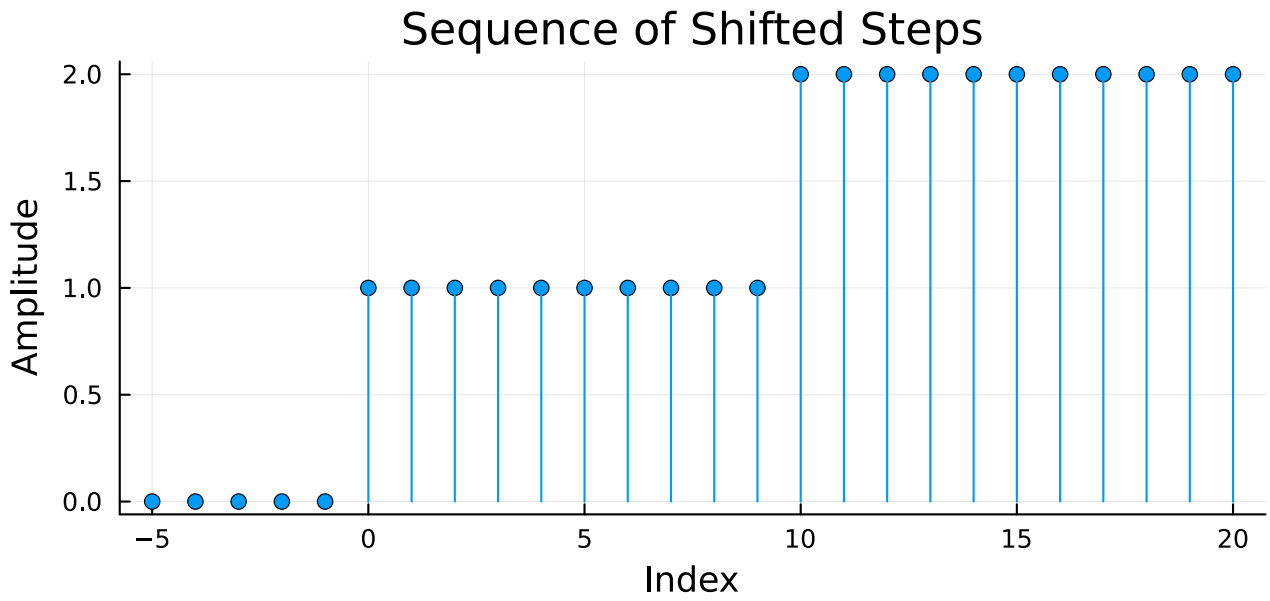

```
begin
    H_fnl = DSPTools.freqz(b_fnl,[1],f_freqz)
    Plots.plot(f_freqz,abs.(H_fnl),leg=false)
    Plots.plot!([0.75,0.75]/2,[0,4],leg=false)
end
```

- The DC Gain o this filter changes as we vary θ_ϵ

```
6.913
```

```
round(real(sum(b_fnl));digits=3)
```

- Working with Sequences




```
begin
    n_t = collect(-5:20)
    x_t = ss.dstep(n_t) .+ ss.dstep(n_t .- 10)
    DSPTools.stem(n_t,x_t;title="Sequence of Shifted Steps")
end
```

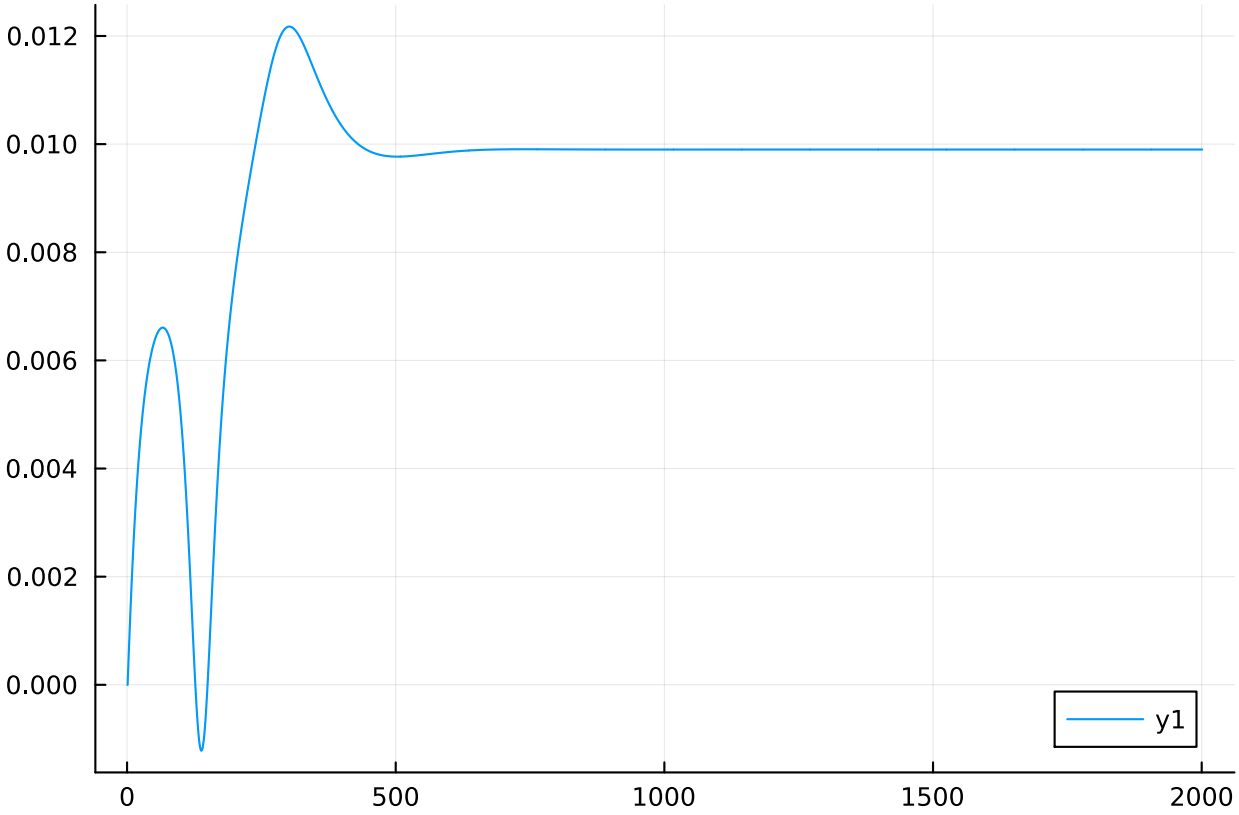
PLL Demo

Change frequency step Δf_{in}: 1000 Hz

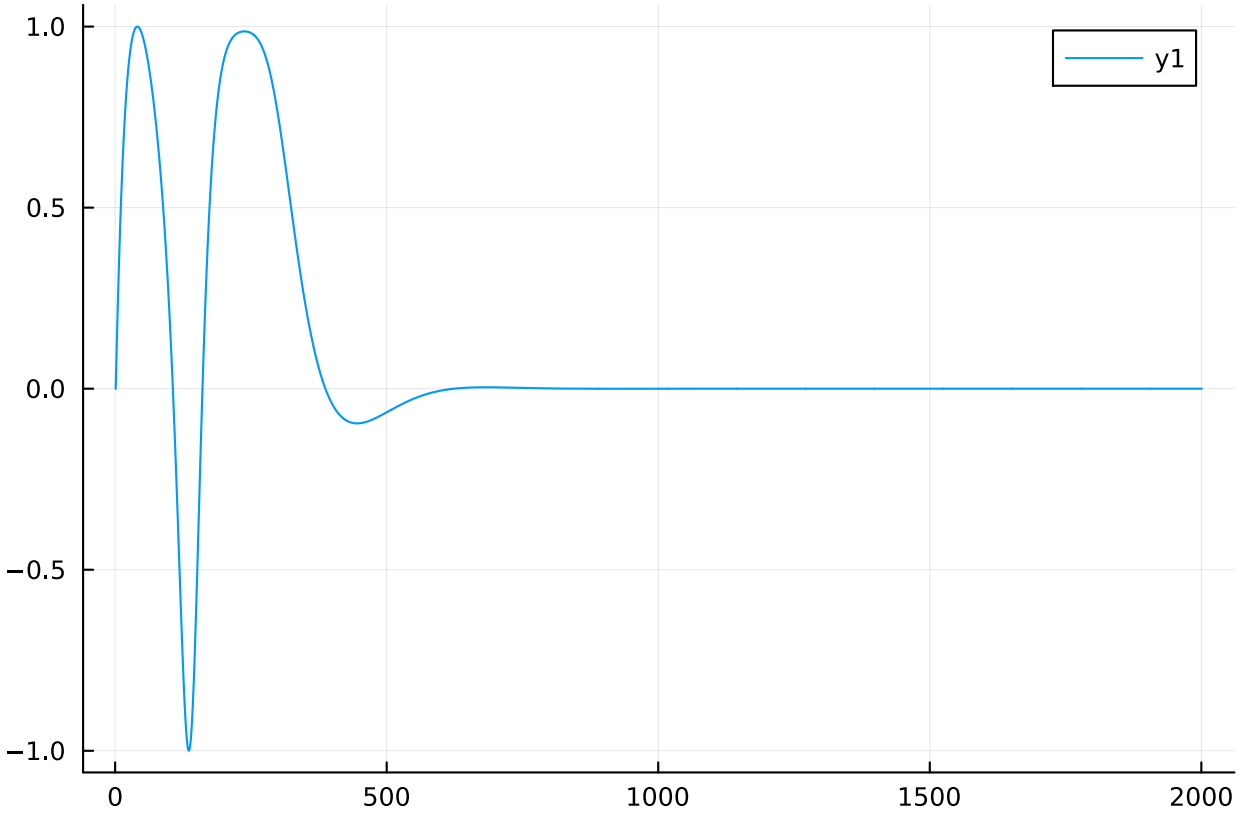
([-0.000628319, 0.0615534, 0.121819, 0.180004, 0.235986, 0.289678, 0.34102, 0.389985, 0.4

```
begin
    f_c = 0.0 # fc_pll=0.0,
    f_clk = 100e3
    n_pll = collect(0:2000)
    x_cbb = exp.(im*2π*D_fin/f_clk*n_pll)
    # x_cbb = exp.(im*2π*1.0e3/f_clk*n_pll)
    y_d_pll, y_lf_pll =
    PLLTools.cbb_PLL(x_cbb,1000.0,1.0;fc_pll=10.0,fclk_pll=f_clk,pll_open=false)
end
```

```
PLL (kp, ki) = (4.272e-03, 5.659e-05)
```




```
begin
    Plots.plot(y_lf_pll)
end
```

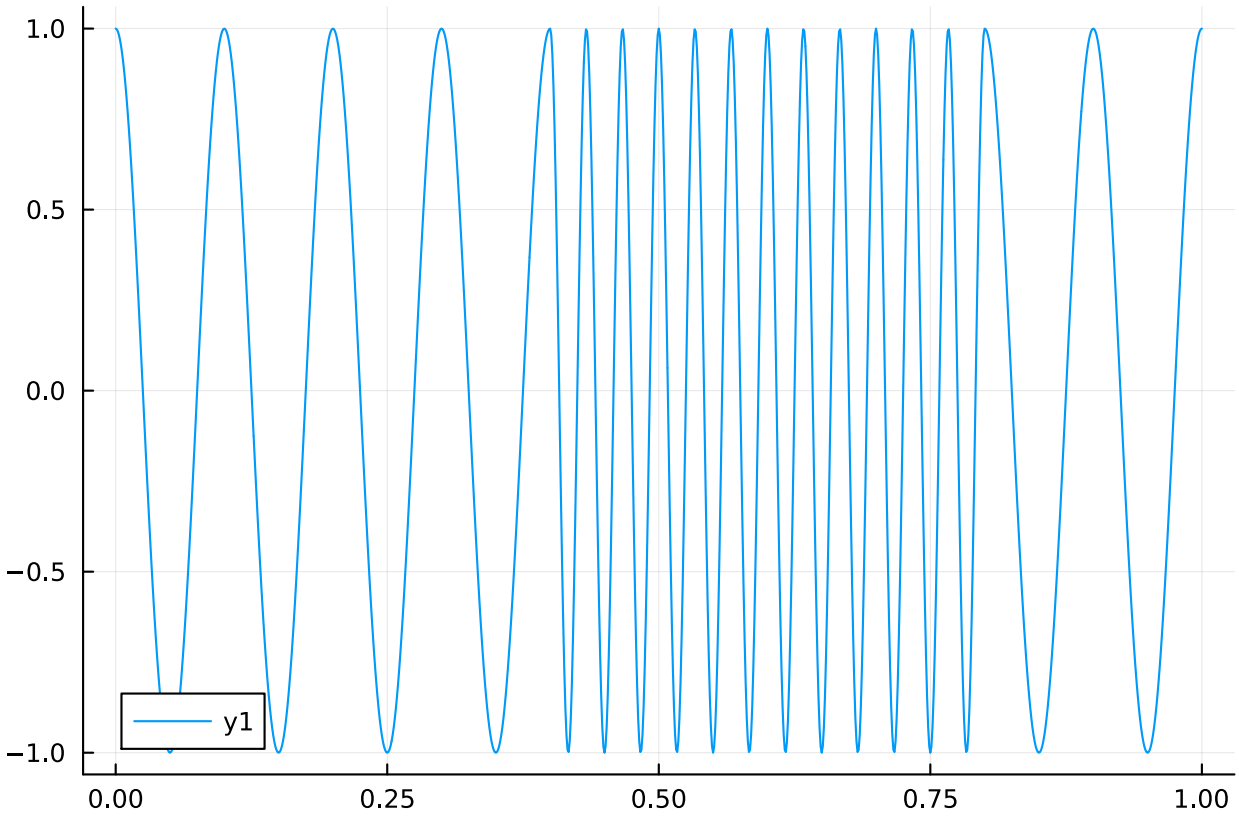



```
begin
    Plots.plot(y_d_pll)
end
```

```
farrow_resample (generic function with 2 methods)
```

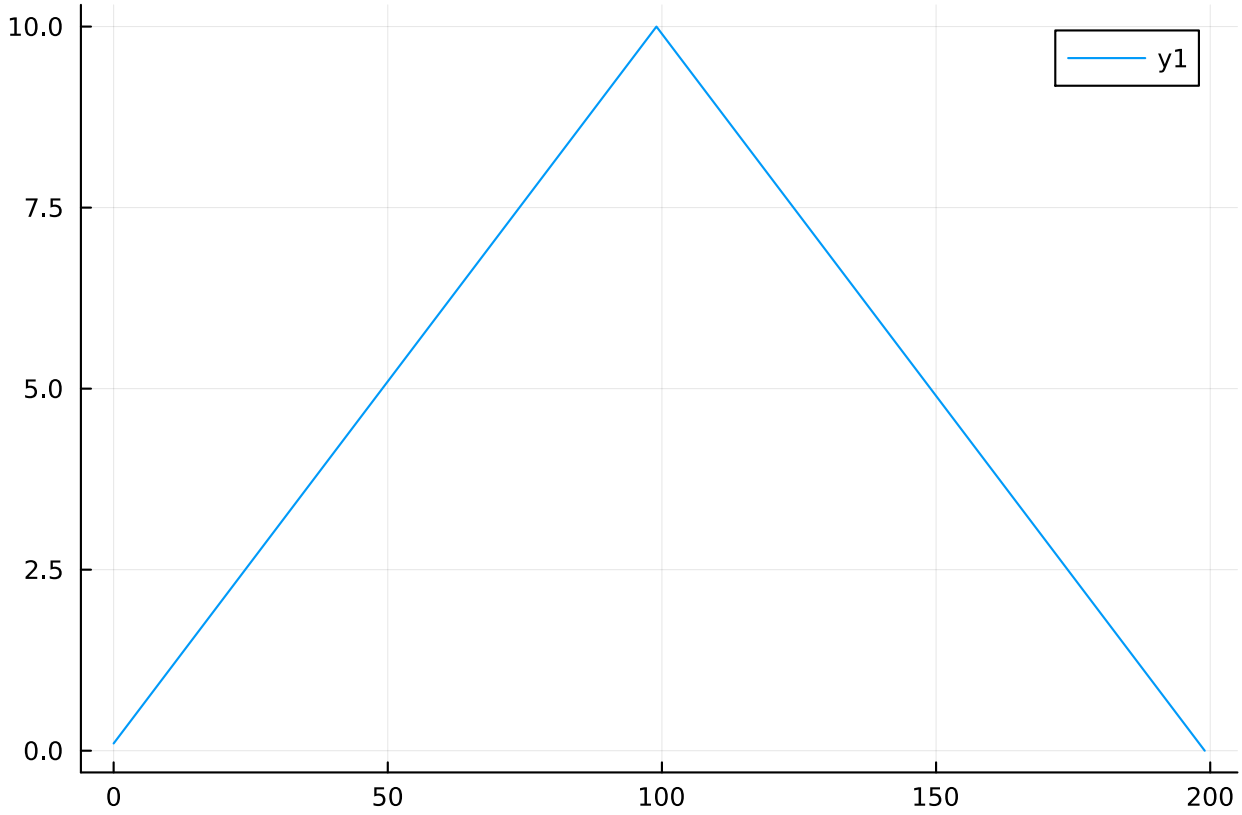
```
CommTools.farrow_resample
```

Waveforms for WebSite Headers




```
begin
    t = collect(0:0.001:1.0)
    f_mod = ss.rect(t .- 0.6,0.4)
    x_fm = cos.(2π * (10 .+ 20*f_mod) .* t)
    Plots.plot(t,x_fm)
end
```

Testing the PLL Accumulator




```
begin
    x = zeros(2*100)
    acc1 = PLLTools.Accum()
    for n in range(0,stop=99)
        PLLTools.accum_update!(acc1,0.1)
        x[n+1] = PLLTools.accum_out(acc1)
    end
    for n in range(100,stop=199)
        PLLTools.accum_update!(acc1,-0.1)
        x[n+1] = PLLTools.accum_out(acc1)
    end
    Plots.plot(collect(0:199),x)
end
```