Julia_NB1

October 16, 2024

1 Basic DSP Modeling using Julia in the Jupyter Notebook

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. When the notebook is first launched (the Julia package `IJulia` needs to be installed first.

[1]:
```
varinfo()
```

[1]:

name	size	summary
Base		Module
Core		Module
Main		Module

1.1 Implement a Frequency Response Calculation

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

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`

[2]:
```

```

[2]:
```
  = 3.1415926535897…
```

[3]:
```
f = collect(0:.001:.5);
H = (1 .+ exp.(-im*2*pi*f))/2; # Note 1*im can be typed as 1im
```

[4]:
```
H
```

[4]:
```
501-element Vector{ComplexF64}:
                     1.0 - 0.0im
  0.9999901304280685 - 0.0031415719827794755im
  0.9999605221019081 - 0.006283019941676304im
```

```
     0.9999111761904045 - 0.009424219857704088im
       0.99984209464165 - 0.01256504772166874im
     0.9997532801828658 - 0.015705379539064146im
     0.9996447363202946 - 0.01884509133496727im
     0.9995164673390624 - 0.02198405915893245im
     0.9993684783030088 - 0.025122159089884778im
     0.9992007750544876 - 0.028259267241012226im
     0.9990133642141358 - 0.03139525976465669im
     0.9988062531806126 - 0.03453001285720289im
     0.9985794501303069 - 0.03766340276396636im

  0.0011937468193873868 - 0.03453001285720303im
  0.0009866357858642205 - 0.03139525976465679im
  0.0007992249455124889 - 0.028259267241012341m
  0.0006315216969912663 - 0.025122159089884831m
  0.0004835326609376467 - 0.021984059158932481m
 0.00035526367970539763 - 0.018845091334967267im
     0.0002467198171342 - 0.015705379539064118im
 0.00015790535835003006 - 0.012565047721668906im
   8.882380959551739e-5 - 0.009424219857704231m
   3.947789809194413e-5 - 0.006283019941676418im
   9.869571931442334e-6 - 0.003141571982779563 6im
                    0.0 - 6.123233995736766e-17im
```

1.1.1 Include a Figure (this figure is used in an interactive Pluto notebook)

1.1.2 Plot $H(f)$

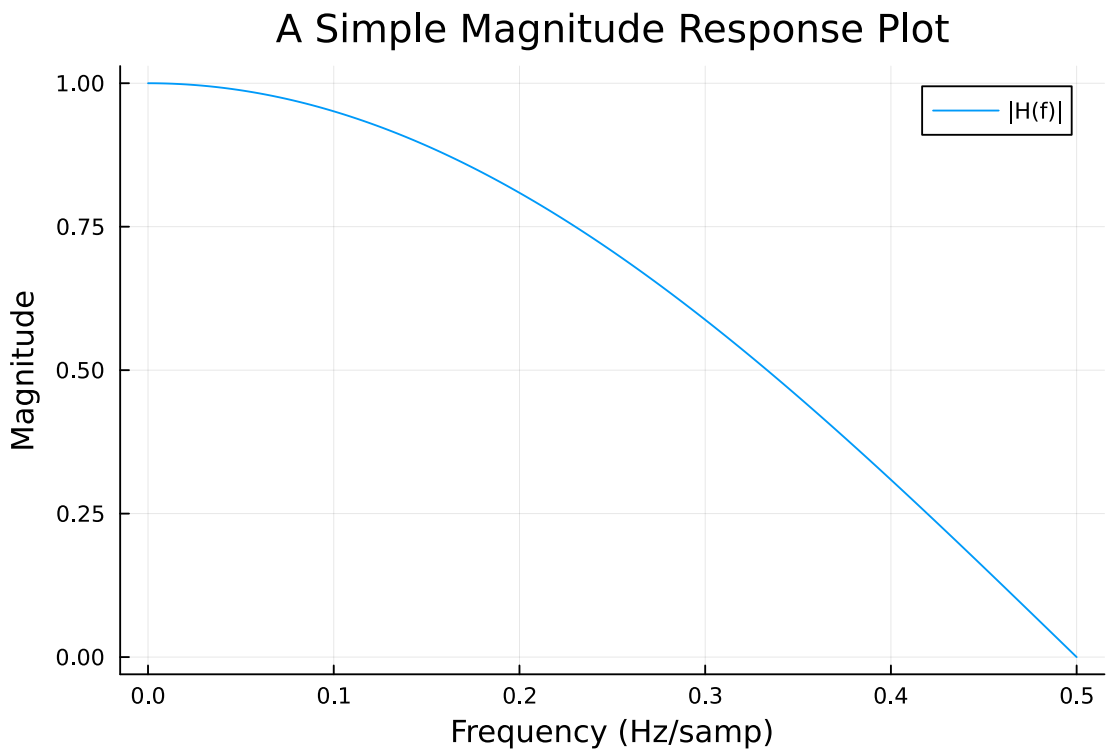
To create plots we need to import a plotting package. There are many such package available, but my preference is to use `Plots`. This package needs to added to the base install of Julia using the `package manager`. Start the Julia REPL (read-execute-print-loop) interface. Once inside the REPL type the 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.

[5]:
```
import Plots
```

[6]:
```
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
```

[6]:



1.2 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.

[7]:
```
"""
    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)
```

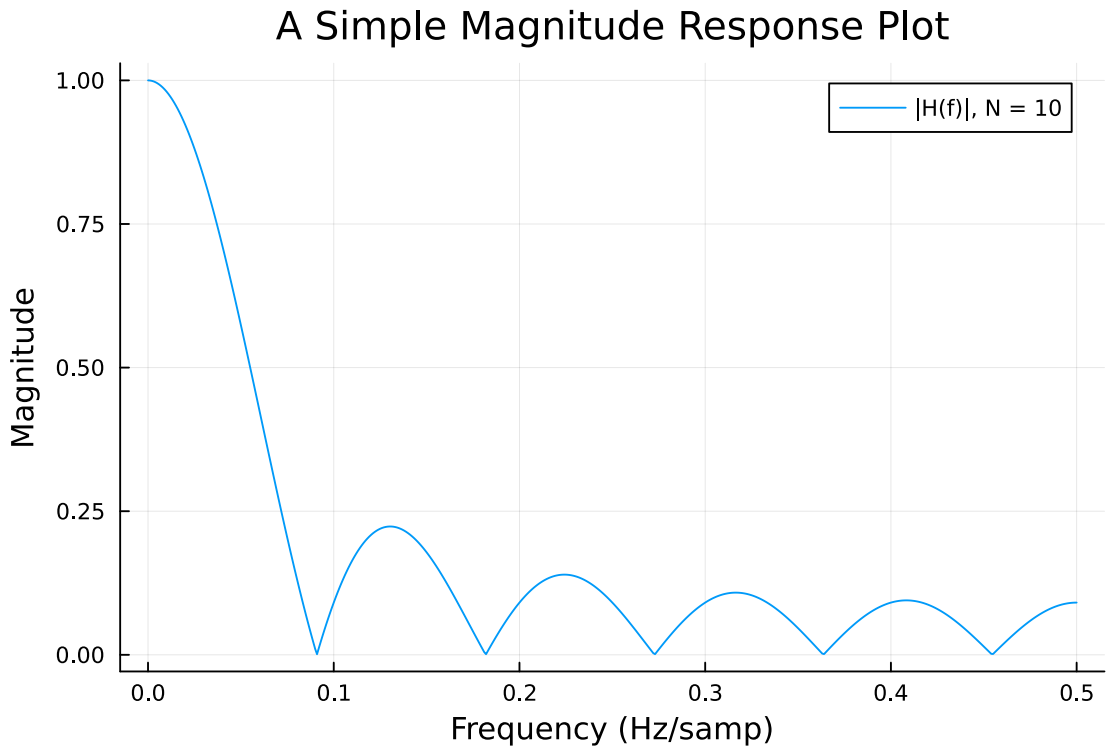
```
    return H_N, f
end
```

[7]:
```
MA_freq_resp
```

[8]:
```
N = 10
H_N, f = MA_freq_resp(N)

Plots.plot(f,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
```

[8]:



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

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

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

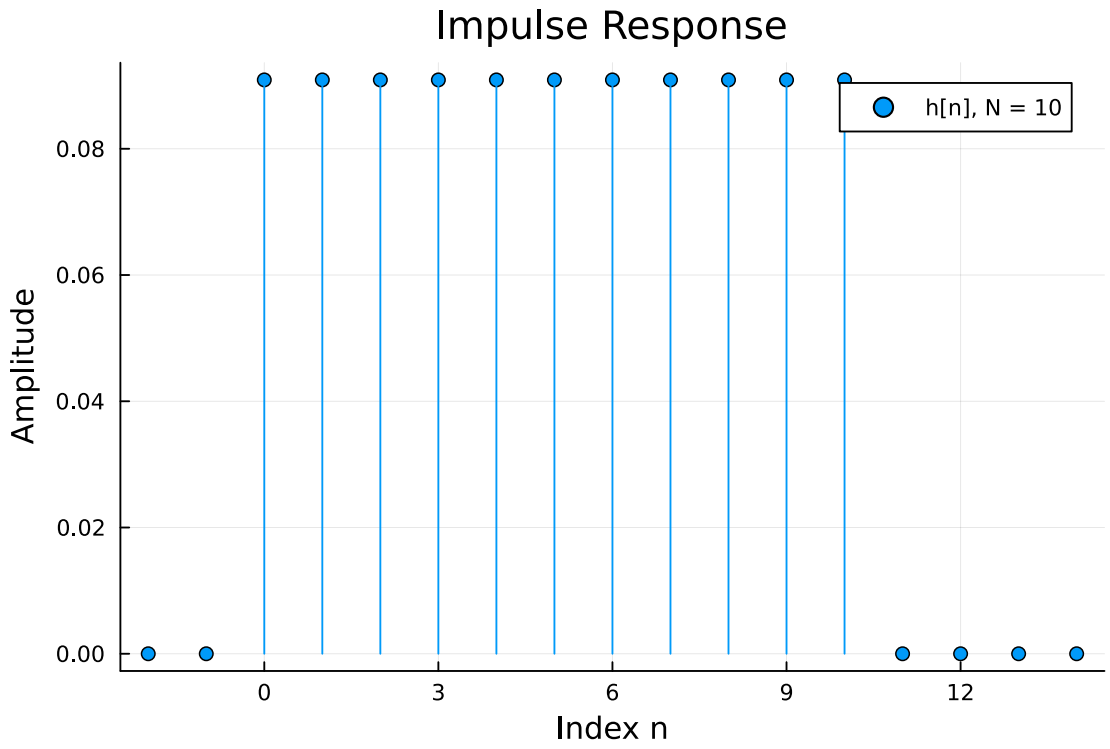
[9]:
```
"""
    h, n = MA_imp(N; n_before=2,n_after=4)
```

```
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
```

[9]:
```
MA_imp
```

[10]:
```
N = 10
h, n = MA_imp(N;)
Plots.plot(n, h, label="h[n], N = $N", line=:stem, marker=:circle,␣
 ↪xlabel="Index n",
           ylabel="Amplitude", title="Impulse Response")
```

[10]:



1.3 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.

To move around you file system, and this case we want to know the path to the `modules` folder relating to where this notebook is placed, we can use the functions `pwd()` and `cd("some_relative_path")`. To get started with importing the custom modules I developed we first `include()` the module `ingredients_function.jl`:

[12]:
```
include("modules/ingredients_function.jl")
```

[12]:
```
ingredients
```

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
```

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

[13]:
```
DSPTools = ingredients("modules/DSPTools.jl")
```

[13]:
```
Main.var"DSPTools.jl"
```

[15]:
```
CommTools = ingredients("modules/CommTools.jl")
```

[15]:
```
Main.var"CommTools.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:

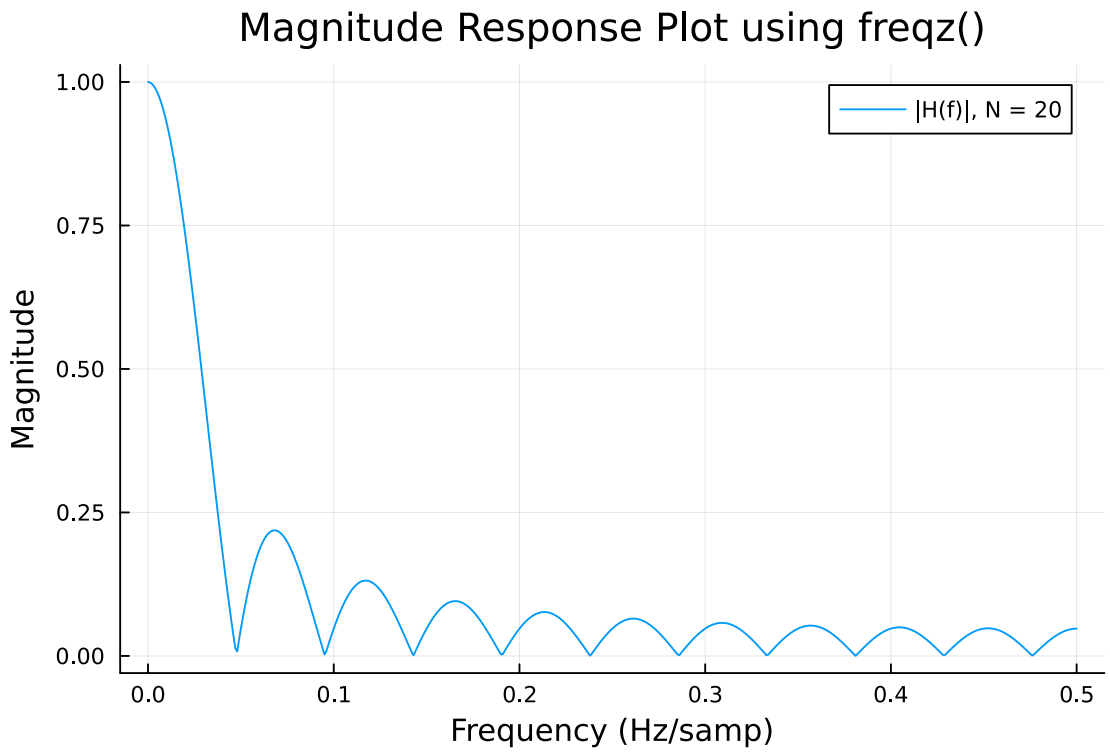
[ ]:
```
DSPTools. # press the tab key after typing the period
```

[ ]:
```
CommTools. # press the tab key after typing the period
```

To see `docstring` help for a given function place the cursor inside the function call and type `shift-tab`

[16]:
```
N = 20
H_N_dsp = DSPTools.freqz(ones(N+1)/(N+1),[1],f)
Plots.plot(f,abs.(H_N_dsp),label="|H(f)|, N = $N",xlabel="Frequency (Hz/
 ↪samp)",ylabel="Magnitude")
Plots.plot!(title="Magnitude Response Plot using freqz()")
```

[16]:



1.4 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* sits on top of a Python install. If you already have ...

[17]:
```
using PyCall
```

[18]:
```
ss = pyimport("sk_dsp_comm.sigsys")
```

[18]:
```
PyObject <module 'sk_dsp_comm.sigsys' from
'/Users/mwickert/.julia/conda/3/x86_64/lib/python3.10/site-
packages/sk_dsp_comm/sigsys.py'>
```

[19]:
```
signal = pyimport("scipy.signal")
```

[19]:
```
PyObject <module 'scipy.signal' from
'/Users/mwickert/.julia/conda/3/x86_64/lib/python3.10/site-
packages/scipy/signal/__init__.py'>
```

[20]:
```
np = pyimport("numpy")
```

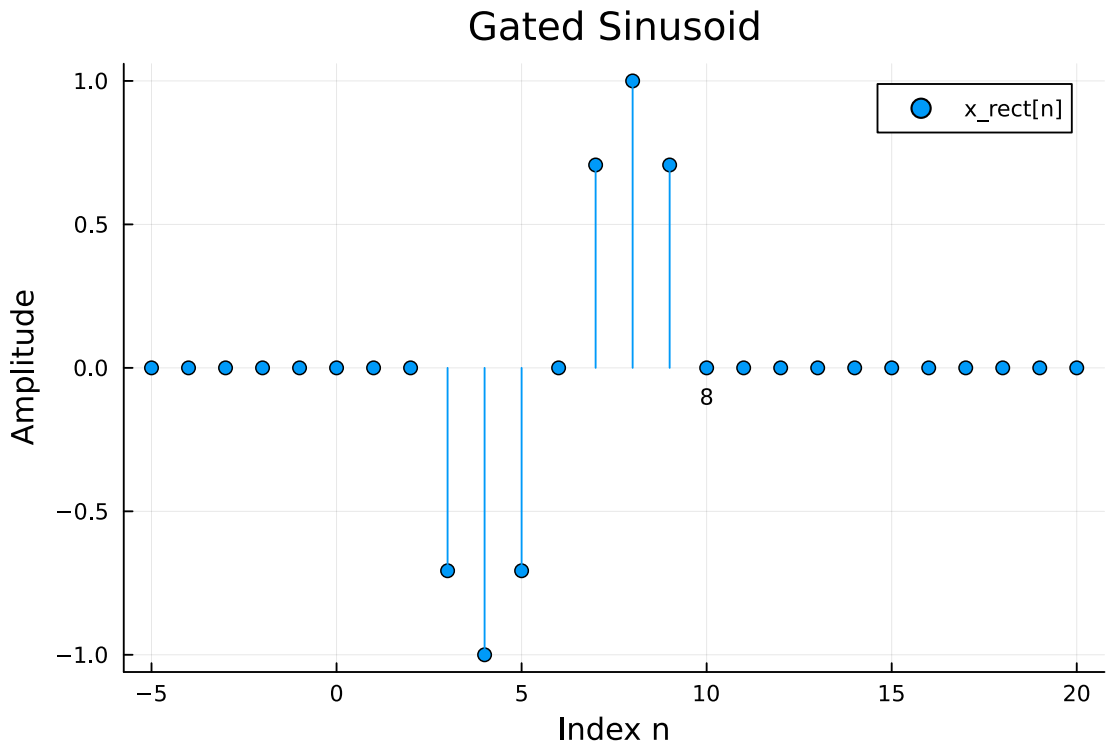
[20]:
```
PyObject <module 'numpy' from
'/Users/mwickert/.julia/conda/3/x86_64/lib/python3.10/site-
packages/numpy/__init__.py'>
```

1.4.1 Using Signal Primitives from `sk_dsp_comm.sigsys`

Create a pulse gated sinusoid using `ss.dstep`

[21]:
```
n_rect = collect(-5:20)
wind_width = 8
wind_start = 2
x_rect = cos.(2*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")
# Add an annotation to the plot
Plots.plot!(annotation=(10, -.1, "$wind_width"),annotationfontsize=8)
```

[21]:



[22]:
```
b, a = signal.butter(5,2*0.1)
```

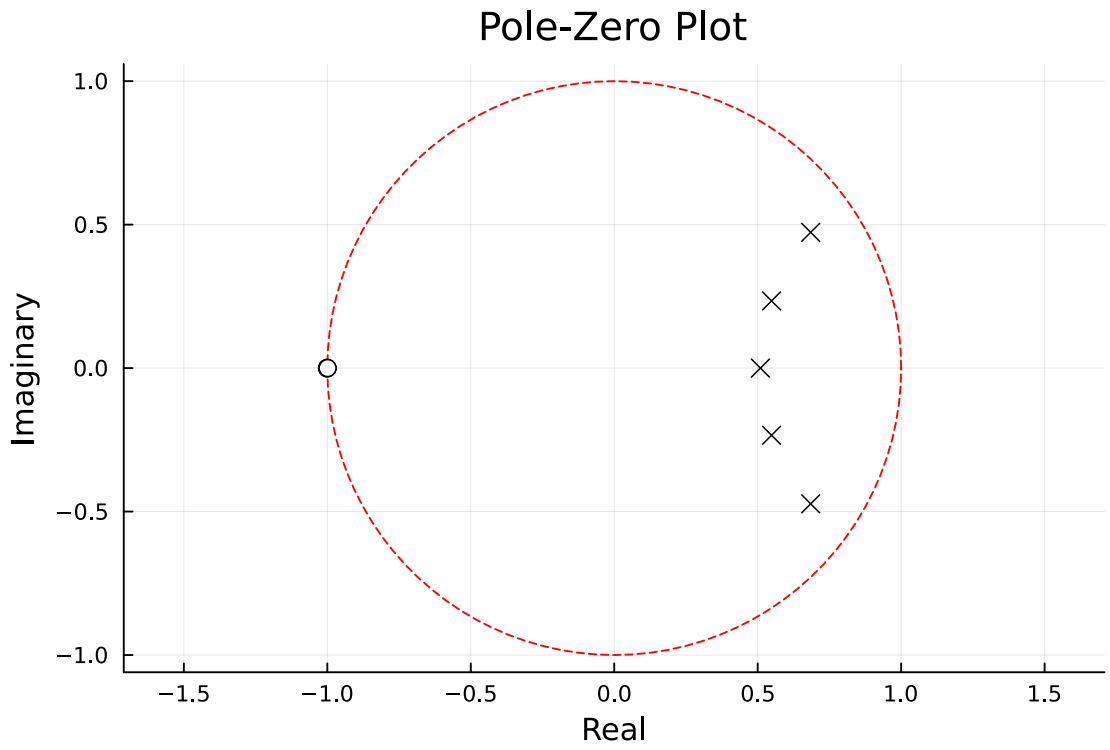
[22]:
```
([0.0012825810789606849, 0.006412905394803424, 0.012825810789606849,
0.012825810789606849, 0.006412905394803424, 0.0012825810789606849], [1.0,
-2.9754221097456828, 3.8060181193204103, -2.5452528683304663,
0.8811300754378365, -0.1254306221553556])
```

[23]:
```
b_roots = DSPTools.roots(b)
```

[23]:
```
5-element Vector{ComplexF64}:
 -1.0007703462305477 - 0.0005593477025354568im
 -1.0007703462305477 + 0.0005593477025354568im
 -0.9997060762559722 - 0.0009060416083107708im
 -0.9997060762559722 + 0.0009060416083107708im
 -0.9990471550269678 + 0.0im
```

[24]:
```
DSPTools.zplane(b,a)
```

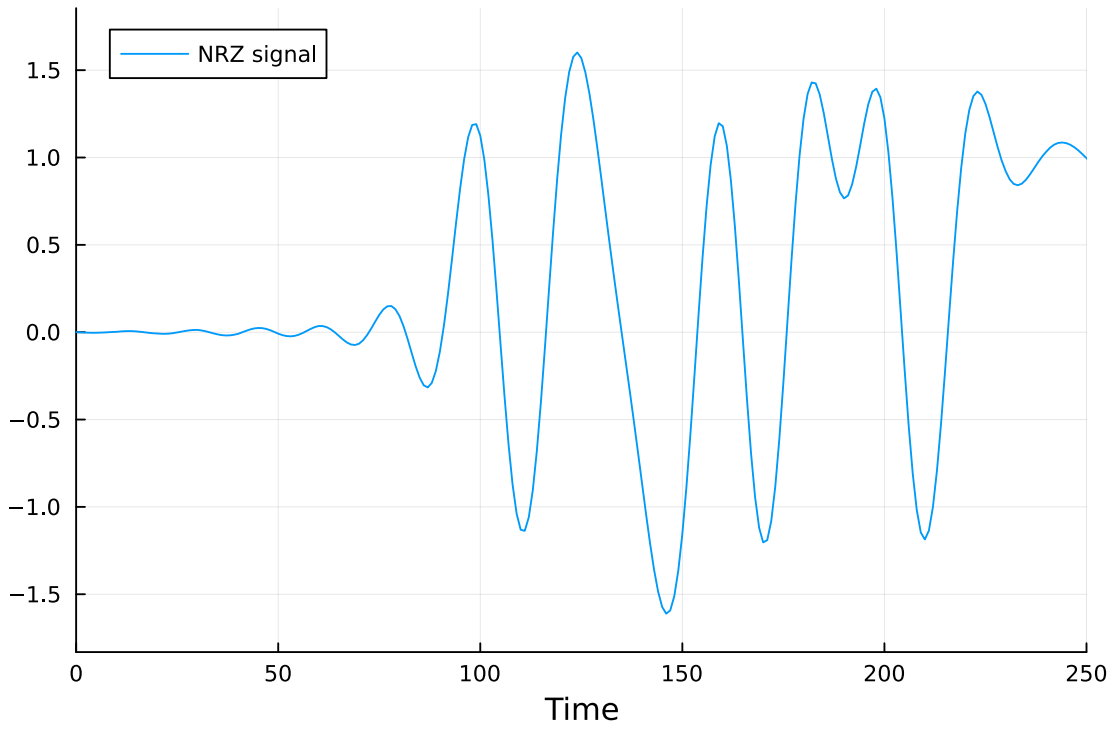
[24]:



[25]:
```
x1, b1, data1 = CommTools.NRZ_bits(500,10,"src";M=10);
```

[26]:
```
Plots.plot(0:length(x1)-1,x1,label="NRZ signal",xlim=(0,250),xlabel="Time")
```

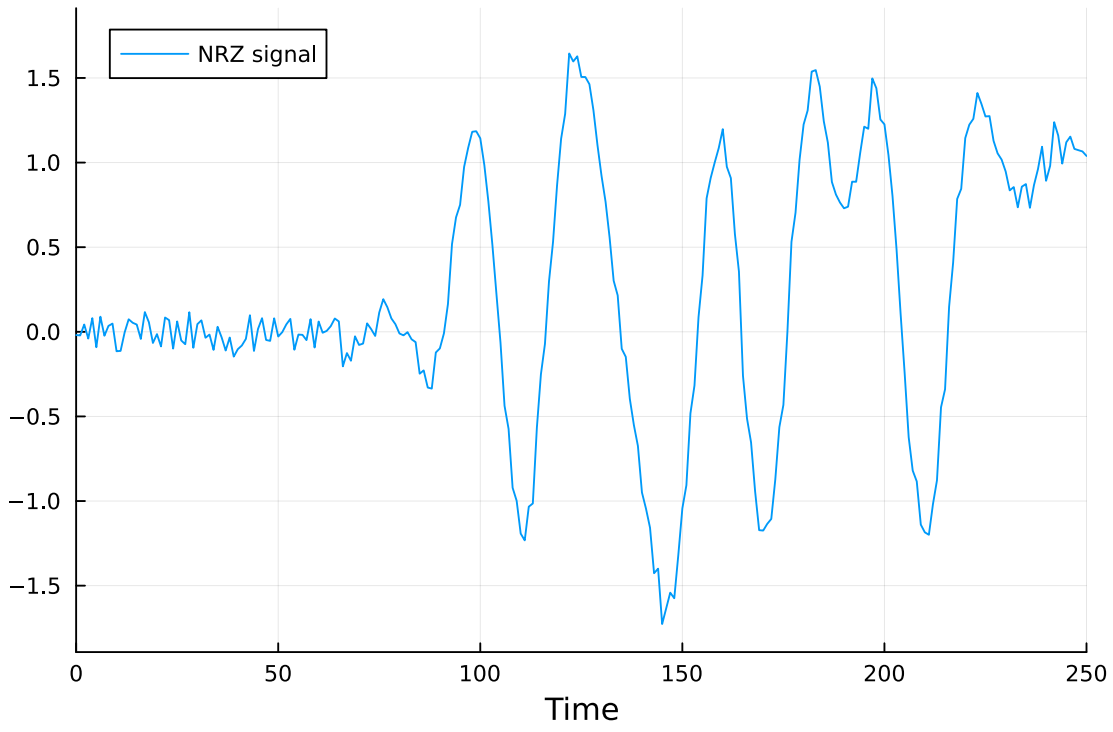
[26]:



[27]:
```
r = CommTools.cpx_AWGN(x1 .+ 0im,30.0,10);
```

[28]:
```
Plots.plot(0:length(r)-1,real(r),label="NRZ signal",xlim=(0,250),xlabel="Time")
```

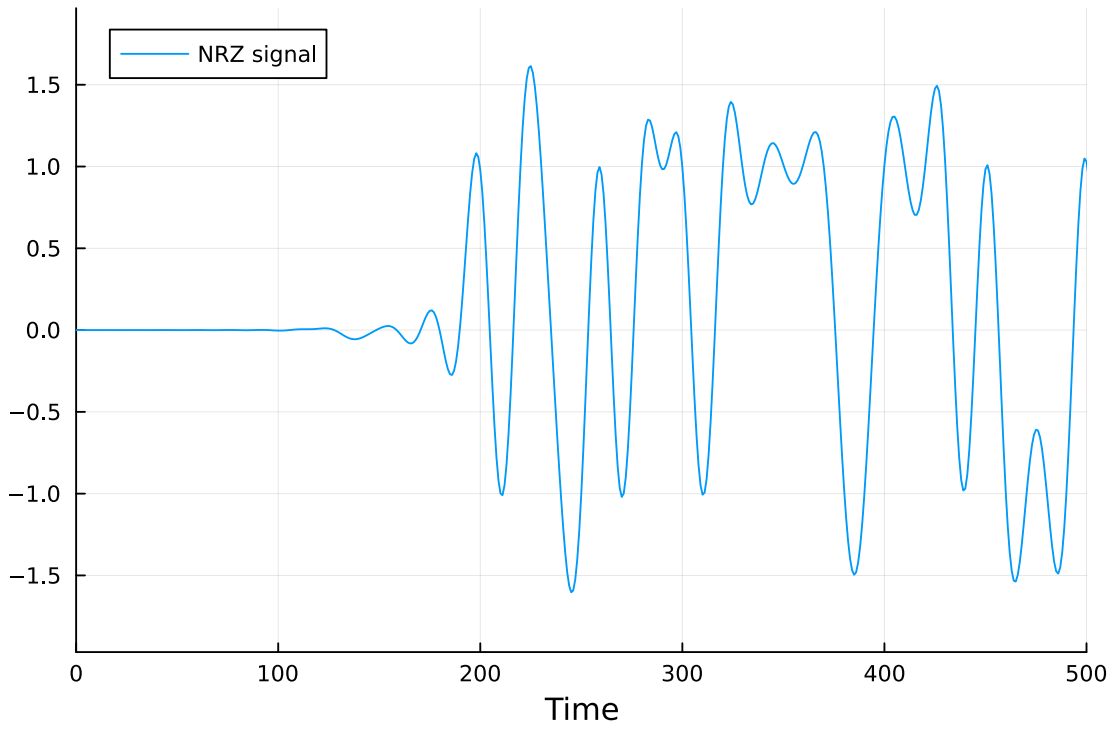
[28]:



[30]:
```
# Match filer the signal r
y = DSPTools.lccde(b1,[1],real(r));
# or just use DSP.filt directly if first using: import DSP
# y = DSP.fi(b1,[1],real(r));
```

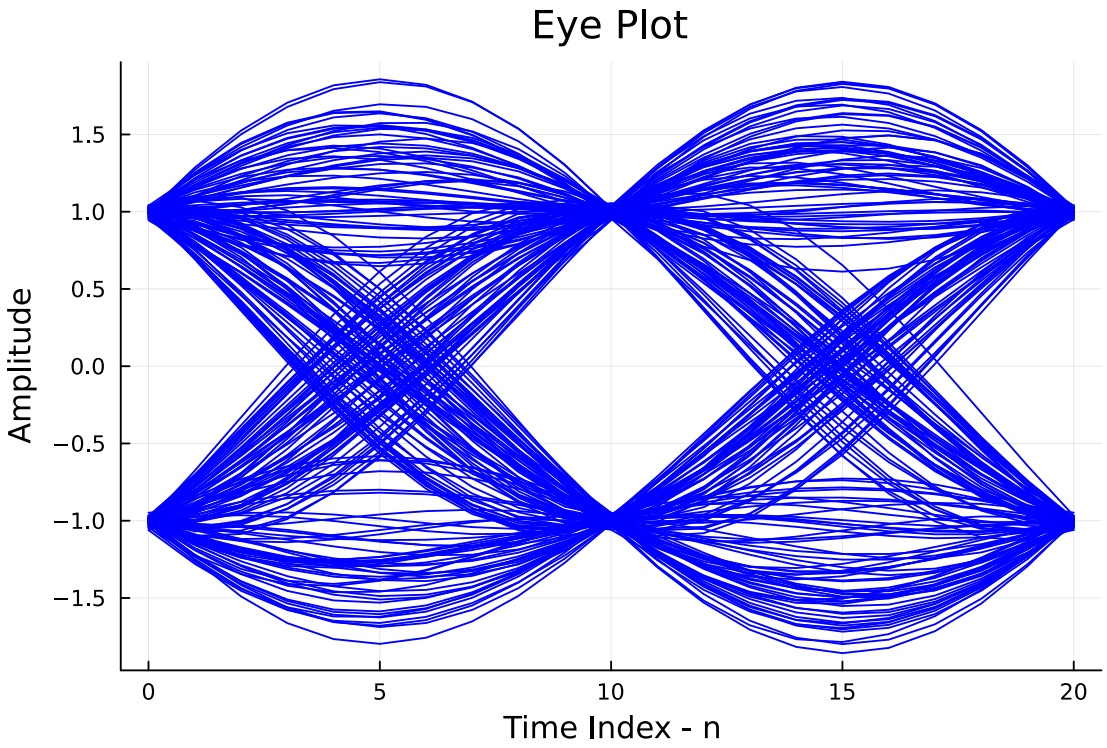
[31]:
```
Plots.plot(0:length(y)-1,y,label="NRZ signal",xlim=(0,500),xlabel="Time")
```

[31]:



[32]:
```
CommTools.eye_plot(real(y),20;S=200+10)
```

[32]:



[34]:
```
using Statistics
```

[35]:
```
var(r)
```

[35]: 0.9915273914726911

[36]:
```
var(y)
```

[36]: 0.9062222309420199

[ ]:
```

```