# Assignment 2 <THMSTE021>JE2.5.X,JE2.6,JE2.7

July 27, 2020

**Brief Remark** Plots.plot() - with plotly() backend - used as default plotting function.

Julia 1.3.1 used in the creation of this notebook.

## 0.1 Julia Exercise 2.5.1 – Visualising Sampled Sinusoid

### 0.1.1 Simulate a sinusoidal signal over enough time to see several cycles.

[1]:
```
using Plots
plotly()
Plots.PlotlyBackend()
```

```
Info: For saving to png with the Plotly backend ORCA has to be installed.
@ Plots /Users/steventhomi/.julia/packages/Plots/5srrj/src/backends.jl:371
```

[1]:
```
Plots.PlotlyBackend()
```

[2]:
```
f01 = 50 #Hz

startTime = -0.1;
stopTime = 0.1;

x(t) = sin(2*pi*f01*t);

plot(x, startTime, stopTime,label=false)
```

### 0.1.2 Generic functions - line plot, scatter plot

[3]:
```
default(size=(600,300)) # Set default Plot canvas size

# default plot that joins samples with straight lines

function linePlot(k)

    fnyquist = 2*f01   # Nyquist sampling rate
    fs1 = k*fnyquist # Define sample rate

    T01 = (1/f01)
```

```
    Δt1 = 1/fs1
    t1 = 0:Δt1:10*T01;

    x(t1) = sin(2*pi*f01*t1); # Create array containing function to be plotted

    plot(t1,x,xaxis=("time [s]"),title = "Line Plot showing the samples␣
 ↪f0=$(f01)Hz fs=$(round(fs1))Hz",label=false);
end
```

[3]: linePlot (generic function with 1 method)

[4]:
```
# scatter plot that joins samples with points

function scatterPlot(k)

    fnyquist = 2*f01  # Nyquist sampling rate
    fs1 = k*fnyquist # Define sample rate

    T01 = (1/f01)

    Δt1 = 1/fs1
    t1 = 0:Δt1:10*T01;

    x(t1) = sin(2*pi*f01*t1); # Create array containing function to be plotted

    Plots.scatter(t1,x,xaxis=("time [s]"),title = "Scatter Plot showing the␣
 ↪samples f0=$(f01)Hz fs=$(round(fs1))Hz",label=false);
end
```

[4]: scatterPlot (generic function with 1 method)

### 0.1.3 Plot the sampled waveforms in order to show the visual effect of sampling at:

### 0.1.4 100x the Nyquist rate

[5]:
```
linePlot(100)
```

[6]:
```
scatterPlot(100)
```

### 0.1.5 10x the Nyquist rate

[7]:
```
linePlot(10)
```

[8]:
```
scatterPlot(10)
```

### 0.1.6 2x the Nyquist rate

[9]:
```
linePlot(2)
```

[10]:
```
scatterPlot(2)
```

### 0.1.7 1.1x the Nyquist rate

[11]:
```
linePlot(1.1)
```

[12]:
```
scatterPlot(1.1)
```

### 0.1.8 On the Nyquist rate

[13]:
```
linePlot(1)
```

[14]:
```
scatterPlot(1)
```

### 0.1.9 0.7x the Nyquist rate

[15]:
```
k = 0.7

linePlot(k)
```

[16]:
```
scatterPlot(k)
```

[17]:
```
print("fs-f0 = ", round(f01*(k*2 - 1)));
```

```
fs-f0 = 20.0
```

### 0.1.10 0.55x the Nyquist rate

[18]:
```
k = 0.55

linePlot(k)
```

[19]:
```
scatterPlot(k)
```

[20]:
```
print("fs-f0 = ", round(f01*(k*2 - 1)));
```

```
fs-f0 = 5.0
```

The visual plots resemble a closer approximation to the actual signal as the sample interval ($\Delta$t) increases. This condition is met when: - fs is greater than 1 and approaches infinity - fs is less than 1 and approaches negative infinity

## 0.2 Julia Exercise 2.5.2 – DFT / FFT Introduction

### 0.2.1 Insert a dft(x) function into Julia and compare it to the fft( ) function.

### 0.2.2 Generic functions - dft

[21]:
```
function dft(x)
    N=length(x)
    X = zeros(N)+im*zeros(N) # Complex array of 0+0im
    for k=1:N
        for n=1:N
            X[k] = X[k] + x[n]*exp(-im*2*pi*(k-1)*(n-1)/N)
        end
    end
    return X
end
```

[21]:
```
dft (generic function with 1 method)
```

[22]:
```
y = [0,1,1,0,0,0,0,0];
```

[23]:
```
using FFTW;
```

[24]:
```
@show fft(y);
```

```
fft(y) = Complex{Float64}[2.0 + 0.0im, 0.7071067811865476 -
1.7071067811865475im, -1.0 - 1.0im, -0.7071067811865476 + 0.2928932188134524im,
0.0 + 0.0im, -0.7071067811865476 - 0.2928932188134524im, -1.0 + 1.0im,
0.7071067811865476 + 1.7071067811865475im]
```

[25]:
```
@show dft(y);
```

```
dft(y) = Complex{Float64}[2.0 + 0.0im, 0.7071067811865477 -
1.7071067811865475im, -0.9999999999999999 - 1.0000000000000002im,
-0.7071067811865477 + 0.2928932188134524im, 0.0 + 1.2246467991473532e-16im,
-0.7071067811865474 - 0.29289321881345254im, -1.0000000000000002 +
0.9999999999999997im, 0.7071067811865469 + 1.7071067811865477im]
```

### 0.2.3 Plot the magnitude and phase

[26]:
```
Y = fft(y);
plot(abs.(Y), lab = "magnitude", title = "Magnitude Plot")
```

[27]:
```
plot(angle.(Y), lab = "phase", title = "Phase Plot")
```

### 0.2.4 Insert a idft(x) function into Julia and compare it to the ifft( ) function.

### 0.2.5 Generic function - idft

[28]:
```
function idft(X)
    N=length(X)
    x = zeros(N)+im*zeros(N) # Complex array of 0+0im
    for n=1:N
        for k=1:N
            x[n] = x[n] + X[k]*exp(im*2*pi*(k-1)*(n-1)/N)
        end
    end
    return (x./N)
end
```

[28]:
```
idft (generic function with 1 method)
```

[29]:
```
@show ifft(Y);
```

```
ifft(Y) = Complex{Float64}[0.0 + 0.0im, 1.0 + 0.0im, 1.0 + 0.0im,
-5.551115123125783e-17 + 0.0im, 0.0 + 0.0im, -5.551115123125783e-17 + 0.0im, 0.0
+ 0.0im, 0.0 + 0.0im]
```

[30]:
```
@show idft(Y);
```

```
idft(Y) = Complex{Float64}[-1.3877787807814457e-17 + 0.0im, 1.0 -
6.938893903907228e-17im, 0.9999999999999999 - 1.942890293094024e-16im,
2.7755575615628914e-17 + 2.7755575615628914e-16im, -9.71445146547012e-17 +
1.1102230246251565e-16im, 1.942890293094024e-16 + 1.3877787807814457e-16im,
5.551115123125783e-17 - 1.3877787807814457e-16im, 9.020562075079397e-16 +
2.220446049250313e-16im]
```

### 0.2.6 Compare the speed of the dft() and fft() functions

### 0.2.7 Generic function - timer

[31]:
```
function timer(N)

    y = randn(N);

    t_dft = @elapsed dft(y);
    t_fft = @elapsed fft(y);

    Δt = t_dft - t_fft;

    println("dft of length $(N) took $(t_dft) seconds");
    println("fft of length $(N) took $(t_fft) seconds");

    println("\nlag between functions lasted $(Δt) seconds");
```

```
end
```

[31]: timer (generic function with 1 method)

### 0.2.8 1024 Samples

[32]:
```
timer(1024)
```

```
dft of length 1024 took 0.036809291 seconds
fft of length 1024 took 0.000139757 seconds

lag between functions lasted 0.036669534000000004 seconds
```

### 0.2.9 4096 Samples

[33]:
```
timer(4096)
```

```
dft of length 4096 took 0.587233114 seconds
fft of length 4096 took 0.000158347 seconds

lag between functions lasted 0.587074767 seconds
```

The time lag is more noticeable with larger array sample sizes

### 0.2.10 Compare the largest power-of-2 size the fft() and dft() functions can compute within 1 second

[34]:
```
N = 1048576
y = randn(N);

t_fft = @elapsed fft(y);

println("fft of length $(N) took $(t_fft) seconds");
```

```
fft of length 1048576 took 0.077071245 seconds
```

The algorithm is highly efficient with values of the 2-to-the-power-of-20 range

[35]:
```
N = 4096
y = randn(N);

t_dft = @elapsed dft(y);

println("dft of length $(N) took $(t_dft) seconds");
```

```
dft of length 4096 took 0.538222898 seconds
```

The fft() is faster than the dft() for powers of 2 as documented above

## 0.3 Julia Exercise 2.5.3 – FFT of a sine wave

### 0.3.1 Time domain: v(t) = 4 cos (20 t) + 2 cos (30 t)

[36]:
```
t = -0.5:0.001:0.5
v(t) = 4*cos(20*pi*t) + 2*cos(30*pi*t)

plot(t,v,title="Time Domain",xaxis="time",label=false)
```

[37]:
```
# convert a function into an array

function functionToArray(T0, Δt)

    i = 0       #step between reads
    count = 0   #array sentinel
    size = 1 + Int(T0/Δt)     #inclusive

    a = zeros(size)

    while i < T0
        a[count+1] = v(i)
        i += Δt;
        count += 1;
    end
    return a
end
```

[37]: functionToArray (generic function with 1 method)

[38]:
```
b = functionToArray(0.2,0.0001)
V = fft(b);

plot(fftshift(abs.(V)),xaxis = ("samples", (975, 1025)), title="Frequency␣
↪Domain",label="magnitude")
```

There is lack of close detail in the magnitude of the frequency domain.

[39]:
```
plot(angle.(V),title="Frequency Domain",xaxis = ("samples", (990,␣
↪1010)),label="phase")
```

### 0.3.2 Applying zero padding in the time domain

[40]:
```
N = length(b)
m = zeros(16*N) # Make array 16x longer.

m[1:N] = b; # Copy x into first N samples. The rest contains zeros.

Y = fft(m);
```

```julia
plot(fftshift(abs.(Y)),title="Frequency Domain",xaxis = ("samples", (14500,␣
 ↪17500)),label="magnitude")
```

Closer detail can be observed in the magnitude of the frequency domain after application of zero padding. Two Sa() functions can be seen at the locations of the two sinusoidal frequencies.

[41]:
```julia
plot(angle.(Y),title="Frequency Domain",xaxis = ("samples", (15500,␣
 ↪16500)),label="phase")
```

## 0.4 Julia Exercise 2.5.4 – Effect of ADC quantization

### 0.4.1 Simulate a sinusoid voltage v = cos.(2*pi*f0*t) that lies in the range: Amin = -0.5 to Amax = 0.5

[42]:
```julia
f0 = 50 #Hz
A = 0.5 #Amplitude

x(t) = A*cos(2*pi*f0*t)

plot(x,-0.02,0.02,title="Voltage",label=false)
```

[43]:
```julia
function functionToArray2(T0, Δt)

    i = 0        #step between reads
    count = 0    #array sentinel
    size = 1 + Int(T0/Δt)     #inclusive

    h = zeros(size)

    while i < T0
        h[count+1] = v(i)
        i += Δt;
        count += 1;
    end
    return h
end
```

[43]:
```
functionToArray2 (generic function with 1 method)
```

### 0.4.2 Quantize the signal into a power-of-2 levels

[44]:
```julia
function signalQuantization(Nbits)

    x(t) = A*cos(2*pi*f0*t)
    m = functionToArray2(0.2, 0.0001);
```

```
    Nlevels = 2^Nbits
    Amax = 1+0.00001 # Add a small amount to prevent problem at extreme
    Amin = -1-0.00001

    m_quantized = (round.( (m .- Amin)/(Amax-Amin)*Nlevels .- 0.5) .+0.5) /␣
↪Nlevels*(Amax-Amin) .+ Amin;

    return fft(m_quantized) # M_QUANTIZED
end
```

[44]: signalQuantization (generic function with 1 method)

### 0.4.3 Number of Bits = 2

[45]:
```
M_QUANTIZED = signalQuantization(2)

plot(abs.(fftshift(M_QUANTIZED)),title="Frequency Domain",label="magnitude")
```

[46]:
```
plot(20*log10.(abs.(fftshift(M_QUANTIZED))),title="Frequency␣
↪Domain(dBv)",label="magnitude") # dBv scale to see wide dynamic range.
```

[47]:
```
plot(abs.(ifft(M_QUANTIZED)),title="Time Domain",label="magnitude")
```

### 0.4.4 Number of Bits = 3

[48]:
```
M_QUANTIZED = signalQuantization(3)

plot(abs.(fftshift(M_QUANTIZED)),title="Frequency Domain",label="magnitude")
```

[49]:
```
plot(20*log10.(abs.(fftshift(M_QUANTIZED))),title="Frequency␣
↪Domain(dBv)",label="magnitude") # dBv scale to see wide dynamic range.
```

[50]:
```
plot(abs.(ifft(M_QUANTIZED)),title="Time Domain",label="magnitude")
```

### 0.4.5 Number of Bits = 4

[51]:
```
M_QUANTIZED = signalQuantization(4)

plot(abs.(fftshift(M_QUANTIZED)),title="Frequency Domain",label="magnitude")
```

[52]:
```
plot(20*log10.(abs.(fftshift(M_QUANTIZED))),title="Frequency␣
↪Domain(dBv)",label="magnitude") # dBv scale to see wide dynamic range.
```

[53]:
```
plot(abs.(ifft(M_QUANTIZED)),title="Time Domain",label="magnitude")
```

The frequency domain experiences a decrease in the magnitude of its lower frequency components with an increase in the number of bits used for quantization process. The time domain experiences a distortion as the number of bits used in the quantization process decrease.

## 0.5 Julia Exercise 2.5.5 – Simulating bandlimited noise

### 0.5.1 Noise Simulation

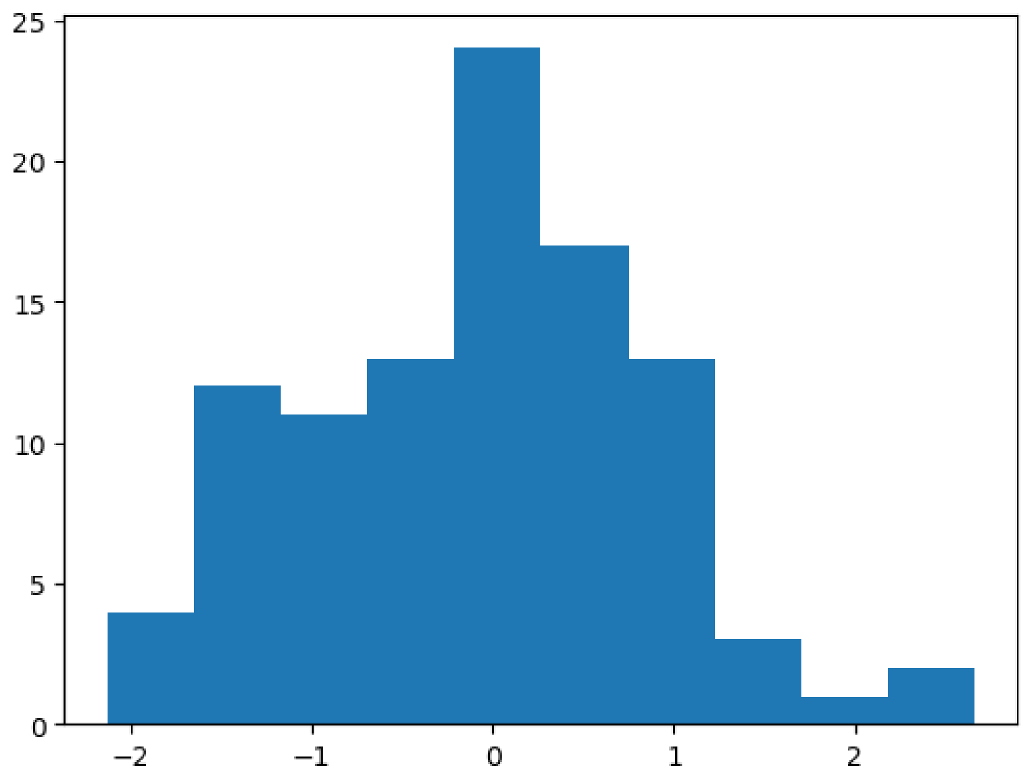
[54]:
```
using PyPlot
```

```
WARNING: using PyPlot.plot in module Main conflicts with an existing identifier.
```

[55]:
```
fs = 1000 # sample rate in Hz
Δt = 1/fs; # sample spacing in s
N = 100 # number of samples
bins = 10 # histogram bins
t = range(0, step=Δt, length=N) # Define time axis
  = 1

randnum =   * randn(N);

PyPlot.hist(randnum,bins); # inspect histogram
```



### 0.5.2 Time and Frequency Domains

[56]:
```
plot(t,randnum,title="Time Domain",label=false) # inspect sampled time domain␣
↪title("Sampled noise, bandwidth B=fs/2")
```

[57]:
```
X = fft(randnum);

plot(abs.(X),title="Frequency Domain",label=false) # inspect DFT frequency␣
↪domain
```

### 0.5.3 Applying frequency-domain zero padding

[58]:
```
pad_factor = 10
Ny = pad_factor * N;

Y = zeros(Ny)+im*zeros(Ny) # Create a complex array of zeros

k_mid = Int(N/2)

Y[1:k_mid]=X[1:k_mid]; # Insert the first half of X

Y[Ny-k_mid+1:Ny]=X[k_mid+1:N]; # Insert the 2nd half of X at the end

plot(abs.(Y),title="Zero-Padded Frequency Domain",label=false) # inspect padded␣
↪array
```

[59]:
```
y = real(ifft(Y)) # Go back to time domain, discard the imaginary components

Ny = length(y);

t_new = range(0, step=Δt/pad_factor, length=Ny) # Define time axis

plot(t_new,y,title="Zero-Padded Time Domain",label=false)
```

[60]:
```
plot(t_new,y,xaxis = ("first 300 samples", (0, 0.03)),title="300 Sample␣
↪Close-Up",label=false) # Plot just first 300 samples
```

### 0.5.4 Bandlimiting Noise using an Ideal LPF

### 0.5.5 Create and display an ideal LPF

[61]:
```
Δ  = 2*pi/(N*Δt) # Sample spacing in freq domain in rad/s
  = 0:Δ :(N-1)*Δ
f =  /(2* )
B = 100 # filter bandwidth in Hz

rect(t) = (abs.(t).<=0.5)*1.0; # rect function definition
```

```
H = rect( /(4* *B)) + rect( (  .- 2* /Δt)/(4* *B) );

plot(f,H,xaxis = ("Frequency in Hz"),title="Ideal Low Pass Filter",label=false)
```

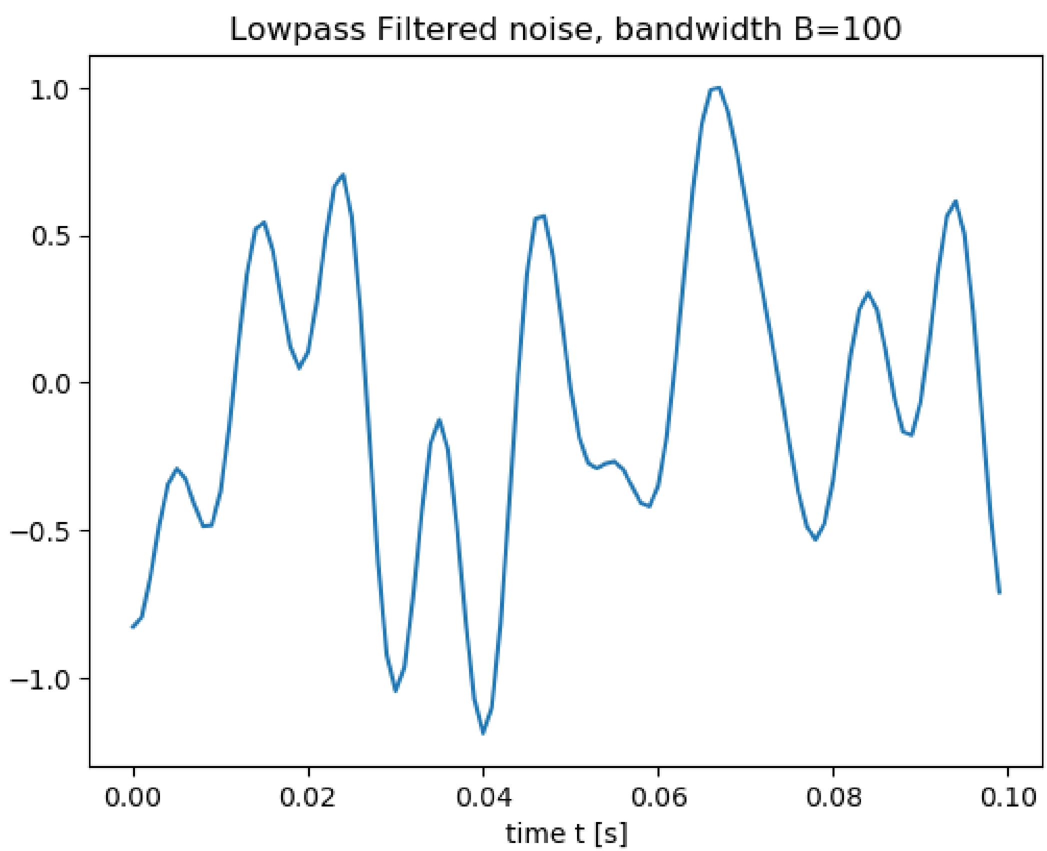
### 0.5.6 Apply Filter to Noise

[62]:
```
X_filtered = X.*H;

plot(abs.(X_filtered),label=false)
```

[63]:
```
x_filtered = ifft(X_filtered)
x_filtered = real(x_filtered)

PyPlot.plot(t,x_filtered)
title("Lowpass Filtered noise, bandwidth B=$(B)")
xlabel("time t [s]");
```




The original time waveform had more spikes (high frequency components) while the bandlimited time waveform is more smooth (absence of high frequency components).

[64]:
```
using Statistics;

 y = std(y)
 xf = std(x_filtered)

println("The original time waveform has  of: ", y);
println("\nThe bandlimited time waveform has  of: ", xf);
```

```
The original time waveform has  of: 0.09460059757910222

The bandlimited time waveform has  of: 0.5091540134349454
```

## 0.6 Julia Exercise 2.5.6 – Discrete fast convolution

[65]:
```
T = 1
t = 0:(T/100):4 # Define time axis

p = rect( (t.-T)/T )
q = rect( (t.-2T)/(2T) )

p1 = plot(t,p,xaxis=("time [s]"),title="Period T",label=false)
p2 = plot(t,q,xaxis=("time [s]"),title="Period 2T",label=false)

xgrid!(t)
plot(p1,p2,layout = (1, 2))
```

[66]:
```
z = ifft(fft(p).*fft(q));

plot(t,real(fftshift(z)),xaxis=("time [s]"),title="Convolution",label=false)
```

## 0.7 Julia Exercise 3.6.1 a) – Discrete fast correlation

### 0.7.1 Produce an auto-correlation function: Rxx(t') = conj(X( ))*X( )

[67]:
```
T = 1;
t = 0:(T/100):4;

s = rect( (t.-T)/T )

x_auotcorr_x = ifft( conj( fft(s) ) .* fft(s) );

plot(t,fftshift(real(x_auotcorr_x)),xaxis=("time [s]"),title="Auto-Correlation␣
 ↪Function",label=false)
```

[85]:
```
A = 1;

y = -A*rect( (t.-T/2)/T) .+ A*rect( (t.-2*T)/(2*T));
```

```
y_auotcorr_y = ifft( conj( fft(y) ) .* fft(y) );

plot(fftshift(real(y_auotcorr_y)),xaxis=("time [s]"),title="Auto-Correlation␣
↪Function",label=false)
```

## 0.8 Julia Exercise 3.7.1 – Matched Filter

### 0.8.1 Define a chirp function.

### 0.8.2 Setting: centre frequency of 10 kHz; bandwidth of 2 kHz; pulse length of 5 ms.

[69]:
```
T = 0.005 # s, pulse length

f0 = 10000 # Hz, centre frequency
B = 2000 # Hz, desired bandwidth of the pulse
K = B/T # Hz/sec, chirp rate

chirp(t) = rect(t/T) * cos(2*pi*t*(f0+(0.5*K*t)));

plot(chirp,xaxis=("time", (0, 0.01)),label=false) # zoom to 0s to 0.01s range
```

### 0.8.3 Define a physically realizable - delayed - chirp function.

[70]:
```
v_tx(t) = chirp(t-T/2)

plot(v_tx,xaxis=("time", (0, 0.01)),label=false)
```

### 0.8.4 Sampled Time Axis

[71]:
```
function nyquistSample(k)

    fnyquist = 2*f0  # Nyquist sampling rate
    fs = k*fnyquist # Define sample rate

    R_max = 10 # metres
    t_max = 2*R_max/c

    Δt = 1/fs
    t_new = 0:Δt:t_max;

    v_tx(t) = chirp(t-T/2); # Create array containing function to be plotted

    plot(t_new,v_tx,title = "Sampled Time Axis waveform",label=false)
end
```

[71]:
```
nyquistSample (generic function with 1 method)
```

```
[72]: c = 343 # m/s

nyquistSample(4)
```

### 0.8.5 Simulated Echo from Target

```
[73]: R = 6 # metres, range of target
A = 1/(R^2); # amplitude decay

t_delay = 2*(R/c); # distance to + from target
Δt = 1/f0 # sample frequency

t = 0:Δt:(T/2+t_delay+T); # range from origin to shift+delay+period

  = T/2+t_delay # cummulative lag

v_rx(t) = A*chirp(t- );

plot(t,v_rx,title="Simulated Echo from Target",label=false)
```

### 0.8.6 Plot time domain waveforms, showing what goes into and out of the matched filter

### 0.8.7 Echo from Target with Additive Noise

```
[74]: v_rnx(t) = randn()+v_rx(t); # echo + additive noise
n(t) = v_rnx(t)-v_rx(t); # noisy echo - echo

plotA = plot(t,n,title="Noise waveform",label=false)
plotB = plot(t,v_rnx,title="Noisy Echo waveform",label=false)

plot(plotA,plotB)
```

A sampling rate of 20x the nyquist rate was used to form a distorted, noisy signal.

### 0.8.8 Apply a matched filter which is created from the reference chirp

```
[75]: plot1 = plot(v_tx,xaxis=("time", (0, 0.025)),title = "Reference␣
 ↪Chirp",label=false) # zoom in to 0s to 0.025s
plot2 = plot(v_rnx,xaxis=("time", (0, 0.01)),title = "Noisy Echo waveform␣
 ↪x(t)",label=false) # zoom in to 0s to 0.01s
plot(plot1,plot2)
```

```
[76]: h(t) = v_tx(-t+t_delay);
plot(h,0.025,0.05,title = "Impulse Response h(t)",label=false) # zoom in to 0.
 ↪025s to 0.05s
```

#### 0.8.9 Generic Functions

[77]:
```
function xToArray(T0, Δt)

    i = 0        #step between reads
    count = 0    #array sentinel
    size = 1 + Int(T0/Δt)     #inclusive

    a = zeros(size)

    while i < T0
        a[count+1] = v_tx(i)
        i += Δt;
        count += 1;
    end
    return a
end
```

[77]:
```
xToArray (generic function with 1 method)
```

[78]:
```
function hToArray(T0, Δt)

    i = 0        #step between reads
    count = 0    #array sentinel
    size = 1 + Int(T0/Δt)     #inclusive

    a = zeros(size)

    while i < T0
        a[count+1] = h(i)
        i += Δt;
        count += 1;
    end
    return a
end
```

[78]:
```
hToArray (generic function with 1 method)
```

The FFTW.fft() and FFTW.iff() functions only take in array data types as formal parameters. It is for this reason that I am forced to convert my functions of time to a closely sampled, accurate array representation.

#### 0.8.10 Plot of output as a function of time

[79]:
```
T0 = 0.04;
Δt = 0.00005;

xarray = xToArray(T0, Δt);
```

```
harray = hToArray(T0, Δt);

taxis = 0:Δt:T0;
yarray = ifft(fft(xarray) .* fft(harray));

plot(taxis,real(yarray),title="Matched Filter Output",label=false)
```

### 0.8.11 Plot the magnitude of the FFT of the pulse, and also of the matched filter.

[80]:
```
X = real(fft(xarray));
H = real(fft(harray));
w = 0:Δt/2*pi:T0/2*pi

plotX = plot(w,fftshift(X),xaxis="frequency [Hz]",title = "FFT of the Pulse␣
 ↪X( )",label=false);
plotH = plot(w,fftshift(H),xaxis="frequency [Hz]",title = "FFT of the Matched␣
 ↪Filter H( )",label=false);

plot(plotX,plotH)
```

Questions: 1. Is the bandwidth of the chirp pulse (as seen in the frequency domain) as expected? 2. What is the shape of the envelope of the output of the matched filter? 3. What happens if you increase the bandwidth? 4. What happens if you increase the length of the pulse?

Solutions: (1) The bandwith is as expected. (2) The envelope peaks at 0.035 seconds, begins at 0.030 seconds and ends at 0.040 seconds. (3) There is no change to the matched filter output. (4) The pulse energy increases, causing the matched filter to easily distinguish between the chirp and its additive noise during filtration.

### 0.8.12 Resolution

[81]:
```
function resolutionN(start,stop)
count = 0;
    while start<stop
        start = start + Δt; # Δt was used to create xarray, and harray above
        count = count + 1;
    end
    return count # total number of elements within given range
end
```

[81]: `resolutionN (generic function with 1 method)`

[82]:
```
start = 0.03; # derived from observation of above plot
stop = 0.04;

numberOfValues = resolutionN(start,stop);
```

```
println("The resolution in seconds is $(1/numberOfValues) seconds");
```

```
The resolution in seconds is 0.005 seconds
```

### 0.8.13 Estimate the Peak SNR at input and at output

[83]:
```
using QuadGK # integration library, integration method in Calculus library is
 ↪deprecated
```

[84]:
```
peakInputSignalPower = v_rnx( )^2; #   is the delay
peakOutputSignalPower = real(yarray[Int(0.035/Δt)])^2; # at t = 0.035
n_in(t) = (v_rnx(t)-v_rx(t))^2;

# quadgk() returns a pair (I,E) of the estimated integral I
# and an estimated upper bound on the absolute error E

noiseAtInput = quadgk(n_in,0,0.01); # integrate the noise^2 from 0 to 0.01

SNR_input = peakInputSignalPower/noiseAtInput[1];

println("Peak SNR at input: ",SNR_input)
```

```
Peak SNR at input: 2.452903614741369
```

[ ]:
```
```