EEE3092F
Signals and Systems II
2022

A/Prof. A.J. Wilkinson

andrew.wilkinson@uct.ac.za

http://www.ee.uct.ac.za

Department of Electrical Engineering

University of Cape Town

Julia Simulation Exercises relating to

Section 2.5 (DFT)

The aim of this exercise is to provide you with an opportunity to

- Simulate time domain waveforms (a sonar chirp pulse with additive noise)
- Use the FFT to inspect waveforms in the frequency domain.

As preparation, you should:

- Read lecture notes: Section 2.5 Discrete Fourier Transform Pair

- Run step by step through the code in the Jupyter notebook:

  Vula → Resources → Software → Julia-sample-code →
  julia_signal_processing_demo.ipynb

Instructions

- These exercises must be done with Julia in a Jupyter notebook.
- Installation instructions for installing Julia and required libraries are inside:
  Vula → Resources → Assignments → Instruction Sheets → Julia Exercises
  1_EEE3092F-Julia_Installation.pdf
- To start Jyputer notebook form the Julia REPL (command line):

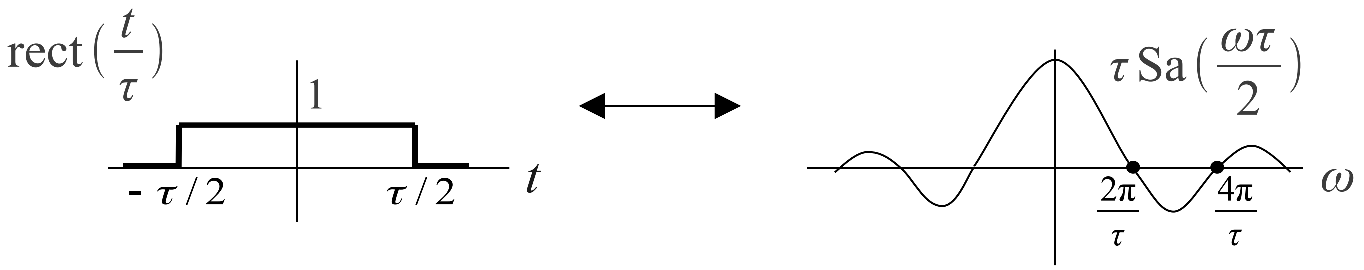
```
using Ijulia <Enter>
noteboook() <Enter>
```

- Sample code:
- Put all you assignment exercises into a single Jupyter notebook file named:
  <student ID>_EEE3092F_Assignment2_Exercises_from_Section_2.5.ipynb
- Add plenty of comments to your code and results.
- For plotting it is recommended to use package: Plots with Plotly backend
  See usage examples in Vula → Resources → Software → Julia-sample-code →
  julia_plotting_with_Plots_and_Plotly_backend.ipynb
  (allows mouse zooming in a notebook; takes ~30s to initialise before first plot.)

Sampling Continuous Signals - Background

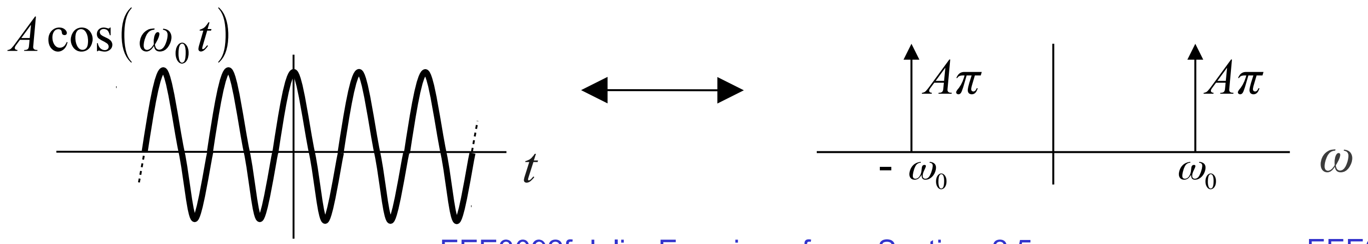
Continuous signals that are bandlimited to bandwidth B Hz can be accurately represented by a sampled sequence if they are bandlimited and sampled above the Nyquist rate of $2B$ samples per second, and with sufficient sample precision. In practice, signals are usually not perfectly bandlimited, resulting in aliasing-related errors. If physical waveforms are sampled by an ADC with finite precision (e.g. 10 bit ADC => 2^10 = 1024 levels), the "quantization error" can be modelled (approximately) as an additional additive white noise process.

Waveforms of finite duration in the time domain are never perfectly bandlimited in the frequency domain e.g.



$\mathrm{rect}\left(\frac{t}{\tau}\right)$
1
$-\tau/2$
$\tau/2$
t
$\tau\,\mathrm{Sa}\left(\frac{\omega\tau}{2}\right)$
$\frac{2\pi}{\tau}$
$\frac{4\pi}{\tau}$
ω

Similarly signals that are strictly bandlimited in the frequency domain will not be of finite extent in the time domain, in which case the sampled signal represents a finite length portion of the signal.



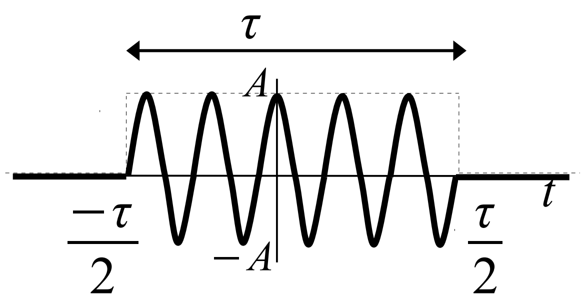
Sampling Continuous Signals - Background

A finite portion of signal $x(t)$ can be modelled as a "windowed" signal being the product of the original signal and a rect() function, e.g.

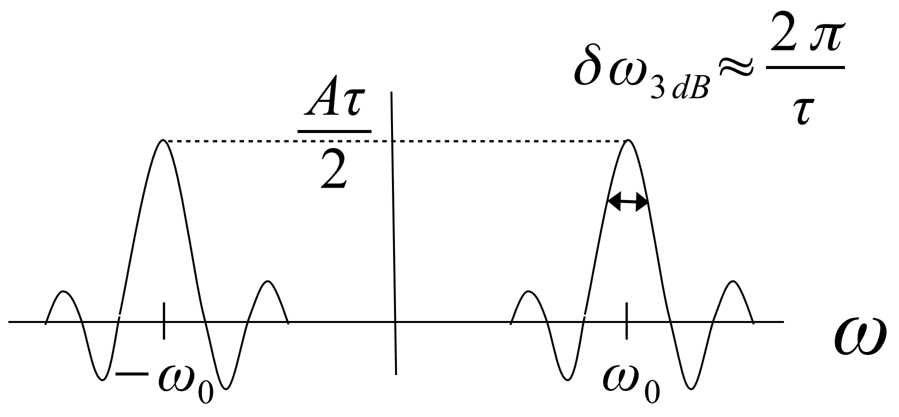
$$x(t)\,\mathrm{rect}\left(\frac{t}{\tau}\right) \Leftrightarrow \frac{1}{2\pi} X(\omega) * \tau\,\mathrm{Sa}\left(\frac{\omega\tau}{2}\right)$$

The effect in the frequency domain is to convolve the spectrum with a Sa() function. Sinusoids that would normally appear as dirac impulses appear as Sa() functions. The sidelobe ringing is of infinite extent, but decays in amplitude.

$$A\cos(\omega_0 t)\,\mathrm{rect}\left(\frac{t}{\tau}\right) \Leftrightarrow \frac{A\tau}{2}\mathrm{Sa}\left((\omega-\omega_0)\tau/2\right)+\frac{A\tau}{2}\mathrm{Sa}\left((\omega+\omega_0)\tau/2\right)$$



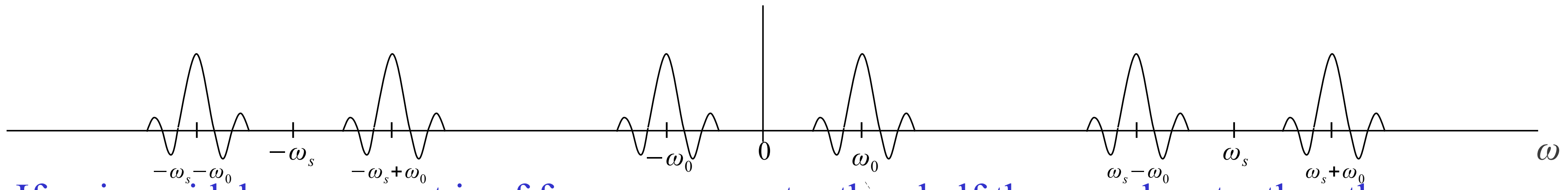
$$\longleftrightarrow$$



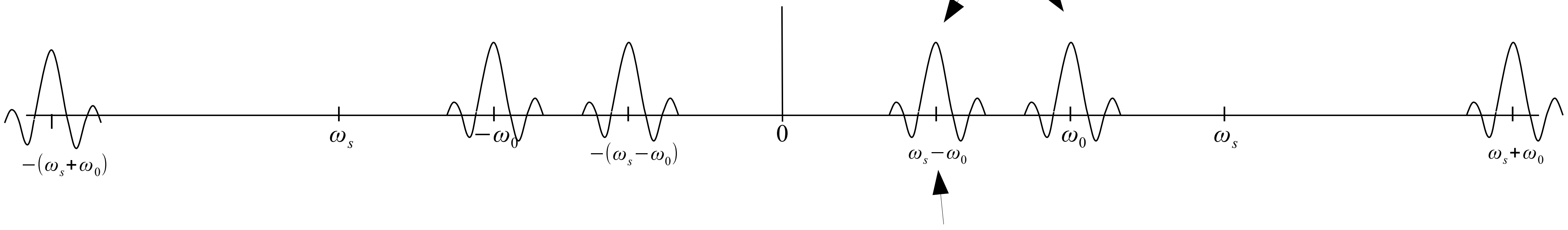
The zero-crossings are at $\omega = \omega_0 + n\dfrac{2\pi}{\tau}$

Sampling Continuous Signals - Background

The act of impulse-sampling a signal at a sample rate of f_s samples per second (or sample spacing $\Delta t=1/f_s$ seconds) creates a periodic spectrum, of period $\omega_s = 2\pi f_s = 2\pi/\Delta t$ rad/s.



If a sinusoidal component is of frequency greater than half the sample rate, then the spectral repeats can appear within the range $-\omega_s/2$ to $\omega_s/2$, an effect known as aliasing. "Aliasing" literally means misidentification of a signal frequency. In the example below, ω_0 is increased (keeping ω_s the same), resulting in the replica at ω_s-ω_0 falling within the $-\omega_s/2$ to $\omega_s/2$ interval. This results in some interesting visual effects in the time domain.



Julia Exercise 2.5.1 – Visualising Sampled Sinusoid

Simulate a sinusoidal signal over enough time to see several cycles. Specify the frequency f_0 Hz of the sine wave, the sample rate f_s in samples per second, start time and stop time. Include about 10 cycles.

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

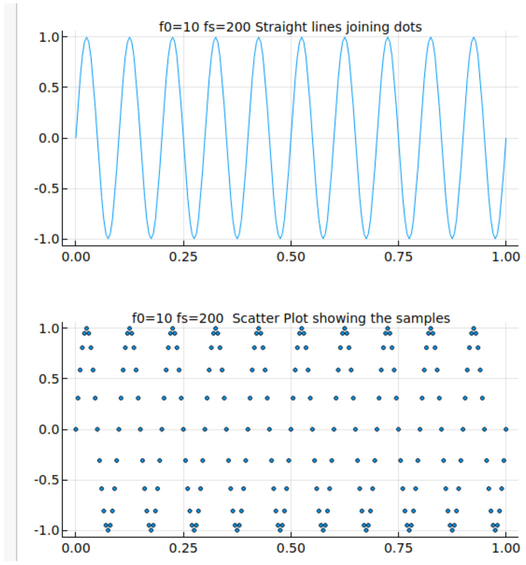
(a) 100x the Nyquist rate.
(b) 10x the Nyquist rate.
(c) 2x the Nyquist rate.
(d) 1.1x the Nyquist rate.
(e) On the Nyquist rate.
(f) Below the Nyquist rate (try 0.7x, and 55x of the Nyquist rate). For these cases, calculate and display the frequency f_s-f_0.

Produce two types of plots:
(i) the default plot that joins samples with straight lines.
(ii) also try plot only the points.

To avoid duplicating code multiple times, you can put your code statements into a function, which can be called, specifying the multiplier k (k=100 or 10 or 2 etc.)

```
function myplot(k)
   …
   plot(...)
end
```

Comment on your visual plots.



Julia Exercise 2.5.1 – sample code with Plots+Plotly

```
using Plots; plotly()    # Plots with Plotly backend allows mouse zooming
default(size=(800,300)); # Plot canvas size
default(label="");       # Turn off legends
default(ticks=:native);  # Ticks on x-axis are labelled properly when zooming in.
function myplot(k)
   f0 =       # Specify the frequency of the sine wave
   T0 = 1/f0  # Calculate the period
   fnyquist = 2*f0   # Calculate Nyquist rate
   fs = k*fnyquist   # Define sample rate
   Δt = 1/fs         # Calculate time step
   t = 0:Δt:10*T0;   # Specify a range of time values
   x = sin.(2*pi*f0*t);    # Create array containing function to be plotted

   # Plot statements using Plots + Plotly backend.
   fig = plot(t,x, title = "Lines joining the dots f0=$(f0) fs=$(fs)")
   display(fig)
   fig = scatter(t,x, markersize=1)
   title!("Scatter Plot showing the samples f0=$(f0) fs=$(fs)")
   display(fig)
end
```

Julia Exercise 2.5.2 – DFT / FFT introduction

The DFT pair of equations are:

N is number of samples

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k e^{j2\pi kn/N} \quad \longleftrightarrow \quad X_k = \sum_{n=0}^{N-1} x_n e^{-j2\pi kn/N}$$

n is time index

k is frequency index

The FFTW library in Julia contains two functions that implement a fast algorithm for the forward and inverse DFT. The functions are fft() and ifft(). "FFT" stands for "Fast Fourier Transform and is an extremely efficient implementation.

Given an array *x* containing *N=length(x)* samples, *X*=fft(*x*) produces an array of *N* frequency domain samples.

The equations above assume the frequency index *k* runs from 0 to *N*-1. In Julia, array indexing starts at 1, i.e. X[1] is the first sample (not X[0], which is not defined). A simple function may be written to perform the DFT using the above formula. For all *k* values (frequency values), evaluate the summation in the equation. Below is a simple function that implements the DFT.

```
function dft(x)
    N=length(x)
    X = zeros(N)+im*zeros(N) # Complex array of 0+0im;  Another method is X=zeros(ComplexF64,N)
    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
```

Julia Exercise 2.5.2 a,b,c continued ...

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

```
x = [0,1,1,0,0,0,0,0]
using FFTW
@show fft(x);
@show dft(x);
```

The complex numbers should be the same (at least to many significant figures).

b) Plot the magnitude and phase via

```
X = fft(x);
using Plots
display( scatter( abs.(X) ) )
display( scatter( angle.(X) ) )
```

c) Try the ifft function.

```
@show invX = ifft(X)
```

Note that even if *x* was purely real, after transforming forward and back again, there may be a small imaginary component. Numerical calculations are not performed to perfect precision.

One can get rid of it via extracting the real part only: real(ifft(X))
(The real() function operates on both scalars and arrays and therefore does not require the "." for arrays.)

Note: you could easily modify the dft() function to implement an idft() function (inverse DFT).

Julia Exercise 2.5.2 d)

d) Compare the speed of the dft() and fft() functions by timing doing some timing tests for different sizes of N.

The fft() algorithm is fast (order NlogN) compared to the direct double for-loop DFT implementation (order N^2).

Timing can be done using either the “@time” macro (which prints how long a function takes to evaluate, as well as memory used), or the “@elapsed” macro which returns the time taken so that it can be stored in a variable.

```
N = 1024
x = randn(N);
t_dft = @elapsed dft(x);
t_fft = @elapsed fft(x);
println("dft of length $(N) took $(t_dft) seconds")
println("fft of length $(N) took $(t_fft) seconds")
```

With timing tests, always repeat the test, because Julia is a “just in time compiler” which means that the first time it calls a function, there will be an additional compiler delay.

The fft algorithm is known to be particularly efficient if the array length is a power of 2.

Compare the speed of the dft vs the fft for different sizes. It is sensible to put the above code in a Julia function `timing_test(N)`.

Try with size N=2^8 = 256, N=2^10 = 1024, N=2^11 = 2028, N=2^12 = 4096. Maybe stop there for the dft().

If you make N too large, the dft() algorithm will take a long time to complete. You may have to restart the julia kernel if it takes too long.

Questions:

(1) What is the largest power-of-2 size fft() that one can compute within 1 second (on your platform)? What is the largest dft() size?

(2) Is the fft() really faster for powers of two? Time the fft for N=2^15=32768 and for N=32767. Which is faster?

Your tests should have shown that the fft algorithm is significantly faster than dft as N increases. Use FFTW’s fft() from now on.

Julia Exercise 2.5.3 – FFT of a sine wave

Load the demo code: Julia-sample-code → julia_signal_processing_demo.ipynb

This code shows how to simulate a sinusoid, and take its FFT.

Go through the code as far as plotting a Fourier spectrum with correctly labelled frequency axes. At the same time read your DFT lecture notes (Section 2.5), so that you understand the steps.

a) Now extract the bits of this code that you need. Modify as required perform a spectral analysis on the signal defined in "Julia Exercise 2.x.4 - Plotting Periodic functions" (which you did in Assignment 1):

$v(t) = 4 \cos(20\pi t) + 2 \cos(30\pi t)$

Show plots of the time domain, and the frequency domain.

You may have to zoom into the frequency domain (i.e. plot a smaller range) to see detail.

b) Try zero padding in the time domain, i.e. extend the length of the time array by adding additional zeros. This will give a finer sample spacing in the frequency domain.

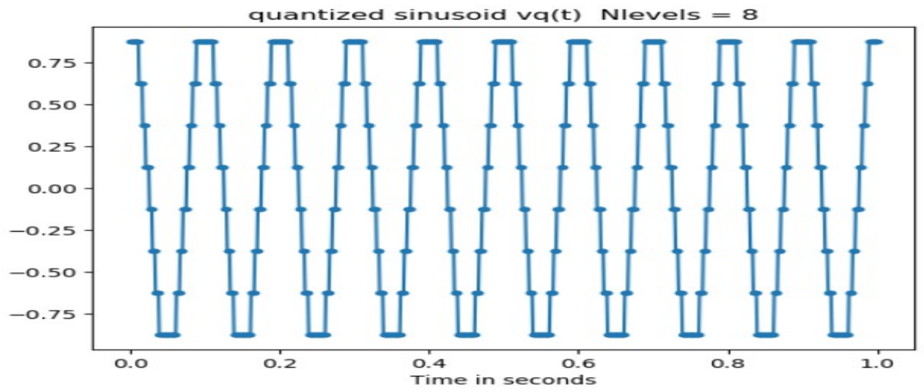
```
N = length(x)
y = zeros(16*N)   # Make array 16x longer.
y[1:N] = x;      # Copy x into first N samples. The rest contains zeros.
Y = fft(y);
display( plot(abs.(Y)) )    # Again, zoom in if required.
```

Question: What do you now clearly see at the locations of the two sinusoidal frequencies?

Julia Exercise 2.5.4 – Effect of ADC quantization

Simulate a sinusoid voltage (as in the previous exercise). v = cos.(2*pi*f0*t)
The signal lies in the range: Amin = -1 to Amax = 1

Quantize the signal into 2^Nbits levels when Nbits is the number of bits of an ADC.



Method:

- Subtract Amin to move into range [0,(Amax-Amin)]
- Divide by (Amax-Amin) to bring into range [0,1]
- Multiply by Nlevels to bring into range [0,Nlevels]
- Subtract 0.5 to bring into range [-0.5,Nlevels-0.5]
- Round to nearest integer number
- Now reverse the process.
- add 0.5, Divide by Nlevels, Multiply by (Amax-Amin), Add Amin

You can use the sample code given on the following page.
Having implemented this, inspect both the unquantized and quantized signal in the time and frequency domains. Increase your sample rate to at least 10 times the Nyquist (or higher) to see the effect.

Question: What effect does quantization have in the frequency domain?

Julia Exercise 2.5.4 – Effect of ADC quantization

```
# Sample code
Nbits = 2  # Try 2, then 3, then 4 etc.

Nlevels = 2^Nbits

Amax = 1+0.00001   # Add a small amount to prevent problem at extremes.

Amin = -1-0.00001

v_quantized = (round.( (v .- Amin)/(Amax-Amin)*Nlevels .- 0.5) .+0.5) /
Nlevels*(Amax-Amin) .+ Amin;

# Display spectrum of the unquantized signal
display( plot(abs.(fft(v)), markershape = :circle, markersize = 1) )

# Display spectrum of the quantized signal
display( plot(abs.(fft(v_quantized)), markershape = :circle, markersize=1) )


# Try a dBv scale to see wide dynamic range.

fig = plot( 20*log10.(abs.(fft(v_quantized))), markershape = :circle,
markersize = 1);
title!("Magnitude of FFT(v_quantized) - dBv scale");
display(fig)
```

Julia Exercise 2.5.5 – Simulating bandlimited noise

We are often interested in simulating both deterministic signals and noise signals.

How do we simulate bandlimited Gaussian noise? In a sampled system, of sample rate f_s the maximum frequency that one can represent is $f_s/2$ Hz. It turns out* that an array filled with independently generated Gaussian random numbers, simulates a sampled Gaussian random process with a flat PSD, i.e. white noise, with a bandwidth of $f_s/2$ Hz. The bandwidth can then be modified by filtering (e.g. by applying a LPF or BPF to give the required bandwidth).

How can we see the continuous waveform that would be between the samples?

Use the zero-padding technique in the frequency domain to effectively increase the sample rate f_s, and then take the ifft. The fft/ifft transforms N samples into N samples. So increasing N via zero padding (inserting zeros into the middle of the DFT array) in the frequency domain produces a finer sample spacing in the time domain since $\Delta t=1/f_s$.

*Theoretical explanation from Lecture Notes Section 3.6 Correlation: The auto-correlation function (ACF) of bandlimited white noise is a Sa() function, with zero-crossings at 1/(2B) seconds. Samples taken from such a signal, spaced at $1/(2B)$ seconds will be uncorrelated. We can create an array of independently generated Gaussian random numbers, corresponding to these samples. If the assumed sample rate is f_s (with time spacing $1/f_s$). Equating $1/(2B) = 1/f_s$, we conclude that the bandwidth of the noise signal is $B = f_s/2$ Hz.

Julia Exercise 2.5.5 a) Generating noise

a) Try the following:

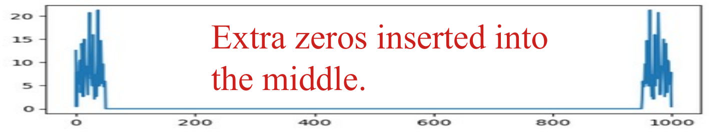
```
fs = 1000  # sample rate in Hz
Δt=1/fs;   # sample spacing in s
N = 100 #  Choose an even number (makes like easier later)
t = range(0, step=Δt, length=N)   # Define time axis
σ = 1
x = σ * randn(N);    # Create the random samples with std dev of σ.
display( histogram(x, nbins=10) ) # inspect histogram
# inspect sampled time domain
display( scatter(t,x, markersize=2, title="Sampled noise, bandwidth
B=fs/2") )
X = fft(x);
display( plot(abs.(X)) )   # inspect DFT frequency domain
```

You should see noisy spectral components across the entire spectrum.

Note: You could estimate the PSD using the method described in the notes on Spectral Analysis. One would need to repeat the above simulation many times and average of the magnitude squared of the spectra. With enough averaging, it should show a flat PSD, i.e. white noise.

Julia Exercise 2.5.5 b) Zero-padding ...

b) Zero-pad in frequency domain to get finer sample spacing in time domain (read lecture notes). Run the code below to inspect the noise waveform resampled at 10x finer sample spacing.



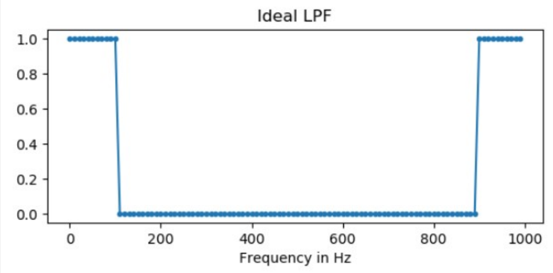
```
pad_factor=10
Ny = pad_factor*N;
Y = zeros(Ny)+im*zeros(Ny)      # Create a complex array of zeros
k_mid = Int(N/2)
Y[1:k_mid]=X[1:k_mid];          # Insert the first half of X
Y[Ny-k_mid+1:Ny]=X[k_mid+1:N]; # Insert the 2nd half of X at the end
plot(abs.(Y));   # inspect padded array
y = ifft(Y);   # Go back to time domain
y = real(y);  # discard the very tiny imaginary components
Ny = length(y)
t_new = range(0, step=Δt/pad_factor, length=Ny) # Define new t-axis
# Plot whole array
display( plot(t_new,y, markershape = :circle, markersize=1) )
# Plot just first 300 samples
display( plot(t_new[1:300],y[1:300], markershape = :circle,
markersize=1) )
```

Julia Exercise 2.5.5 c) Bandlimiting with a LPF ...

c) Now try bandlimiting the noise using an ideal LPF. First create and display the filter. Then apply to the noise waveform in the frequency domain. Here the bandwidth B=100Hz. The original bandwidth was fs/2=1000/2=500Hz.

(Read the lecture notes Section 2.5, showing the form of a LPF in the DFT domain).

```
# Create a Filter transfer function
Δω = 2*pi/(N*Δt)   # Sample spacing in freq domain in rad/s
ω = 0:Δω:(N-1)*Δω
f = ω/(2*π)
B = 100 # filter bandwidth in Hz
# In the sampled frequency domain. add a rect centred on zero to one centred
at the next repeat
# i.e. centred on 0 rad/s an on 2pi/Δt rad/s.
rect(t) = (abs.(t).<=0.5)*1.0;
H = rect(ω/(4*π*B)) + rect( (ω .- 2*π/Δt)/(4*π*B) );
# Note, H in this case is purely real.
fig = plot(f,H,, markershape = :circle, markersize=1);
Title!("Ideal LPF")
Xlabel!("Frequency in Hz")
display(fig)
```



Julia Exercise 2.5.5 c) continued ...

Apply filter to noise:

```
X_filtered = X.*H;
display( plot(abs.(X_filtered)) )


x_filtered = ifft(X_filtered)
x_filtered = real(x_filtered)
fig = plot(t,x_filtered)
title!("Lowpass Filtered noise, bandwidth B=$(B)")
xlabel!("time t [s]")
display(fig)
```

Note: compare this bandlimited noise to the original noise waveform (part a) in the time domain. There is some discussion on this in Section 2.4 on noise.

How do the waveforms differ?
Calculate the standard deviations of each as a measure of spread.

Julia Exercise 2.5.6 – Discrete fast convolution

Discrete convolution is discussed in the lecture notes. To see the result of convolving sequence x with sequence y, a quick way is to use the technique known as "fast discrete convolution" using the fft:

```
x = <some waveform>
y = <another waveform>
z = ifft(fft(x).*fft(y));
```

Note: the result is a circular convolution, equivalent to convolving x with a "periodic extension" of y (see lecture notes). The result is a periodically repeating time sequence. Only one period is stored in the z array. If the result would normally be centred on zero time, then for the discrete case, the first part of the array will show half the result (i.e. t >= 0), and the end of the array will show what would normally be the part for t<0. This happens because the end of the array contains part of the next periodic repeat.

Try the discrete convolution of

`x = rect( (t.-T)/T )` where `rect(t) = (abs.(t).<=0.5)*1.0;`

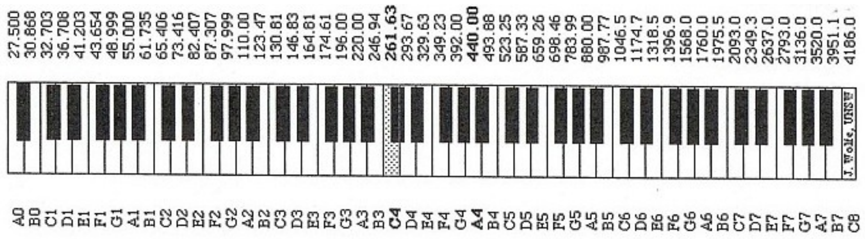
`y = rect( (t.-2T)/(2T) )`

The rect() pulses have been delayed by their pulse width so that they start after time zero. Specify `t=0:dt:tmax` with a suitable time step and length. Plot the result (discarding any small imaginary component resulting from finite precision – we know that the result should be real).

```
plot(real(z))
```

For continuous signals, this would result in an up-sloping rise, a horizontal portion and a down-sloping portion. To put "zero time" in the middle, use the `fftshift()`, I.e. `plot( fftshift(z) )`

Julia Exercise 2.5.7 – Spectral Analysis of Music



Write julia code to
- load an audio “wav” file
- extract a specified portion [t1,t2]
- display in time domain
- display in frequency domain
(You should be aware that a 5 min recording at 44100 Hz creates 13,230,000 samples. Limit your plots to shorter sections (say 10s) to avoid memory problems with Plots/plotly, which creates large plot objects.)

Apply your code to three examples:
(1) Piano music: Beethoven - Für Elise (first 10 seconds)
Questions: Inspect the FFT. You should see the distinct piano notes in the spectrum. Determine the frequencies of the highest three peaks. Determine which piano notes do these frequencies correspond to?
What are the frequencies of the highest and lowest notes played?

(2) Master KG - Jerusalema [Feat. Nomcebo] (first 10 seconds)
Questions: Inspect the time domain and determine the period of the drum beat in seconds.
How does the (dominant) drum beat appear in the frequency spectrum? Describe its features.
How can you determine the fundamental frequency of the beat from inspection of the frequency spectrum?

(3) A short recording of your own voice singing the chorus of Jerusalema (a capella).
https://lyricstranslate.com/en/jerusalema-jerusalem.html-0
Questions: Inspect the time domain and frequency spectrum. Describe significant features in time and frequency domains.
How does the spectrum of your recording compare to (2)?

Julia Exercise 2.5.7 – sample code

You may need to convert your audio files from mp3 to wav.

The free audio editing software Audacity https://www.audacityteam.org/https://www.audacityteam.org/ can load many audio formats and export as a wav file. Audacity can also record voice audio for your rendition.

The Julia library WAV.jl provides functions to read, write, and append to audio WAV files (functions wavread(), wavwrite() and wavappend()). The function wavplay() provides audio playback.
In Julia, install WAV via: `using Pkg; Pkg.add("WAV")`
Visit https://github.com/dancasimiro/WAV.jl and run the simple example.

Here is some Julia code to get you going.

```
using WAV, FFTW, Plots;
plotly(); default(ticks= :native)

data, fs = wavread("song.wav") # Read stereo wave file
# wavread() returns 2D data array (matrix) and also the sample rate fs.
# If stereo, the 1st column holds one channel and the 2nd column holds the other.
@show Nsamples, Nchannels = size(data); # show size of the array; if stereo, it
will be a 2D array.
println("Sample rate fs = ",fs)
wavplay(data, fs)   # play the sound(turn up your volume!).
# To abort playing, press ctrl-C a few times.
```

Julia Exercise 2.5.7 – sample code

```
# Music is usually recorded in stereo so data is a 2D array.
channel1 = data[:,1]  # extract 1st column (left channel I think)
# channel2 = data[:,2]  # if stereo, then there is a 2nd channel in the 2nd column.
dt = 1/fs  # calculate sample spacing from sample rate
t_axis_all = (0:length(channel1)-1)*dt   # time axis for plotting
# If you want to plot the entire file plot(t_axis_all, ch1) using plotly() there will be
memory problems with a large array of millions of elements.
# To avoid this problem, plot every 10th sample via specifying a range with a step of 10.
fig = plot(t_axis_all[1:10:Nsamples], channel1[1:10:Nsamples]);
title!("Channel1 - every 10th sample", xlabel="Time [seconds]");
display(fig)
# Now extract a portion of sound from time t1 to t2 seconds
t1 =0;   t2=10;  # specify start time t1 and end time t2
i1 = Int(round(t1/dt))+1  # calculate index of sample closest to t1
i2 = Int(round(t2/dt))+1  # calculate index of sample closest to t2
y = channel1[i1:i2]     # extract relevant portion from channel1
@show N = length(y)     # Get size of array
wavplay(y, fs)   # play the sound; to abort press ctrl-C a few times.
t=(0:N-1)*dt;    # Define a time axis for the extracted portion
display( plot(y, xlabel="Sample number") )    # Plot array vs index number
display( plot(t,y, xlabel="Time [seconds]") )  # Plot array vs time

# Next calculate the  fft(y)
# display abs.() of fft.  Label frequenct axes in Hz etc.
# For visualisation of a relevant portion use plot(array[index1:index2])
```

EEE3092F
Signals and Systems II

End of handout