Assignment 3 <THMSTE021>JE6.1,JE6.2,JE6.3

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 6.1 – DSB-SC AM

0.1.1 Julia Exercise 6.1a – DSB-SC Modulation

0.1.2 Simulate double sideband suppressed carrier amplitude modulation (DSB-SC AM) with the following parameters:

0.1.3 Modulating waveform is an audio signal: f(t) = cos(2 fmt) where fm=1 kHz.

0.1.4 Carrier wave oscillator: cos(2 fct) where fc=20 kHz.

0.1.5 Modulated carrier wave signal: (t) = f(t) cos(2 fct)

[1]:
```julia
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()
```

0.1.6 (i) Plot the modulating waveform

[2]:
```julia
fmi = 1000 #Hz

startTime = -0.0015;
stopTime = 0.0025;
Δt = (stopTime-startTime)/1000;

ti = startTime:Δt:stopTime;

xi(t) = cos(2*pi*fmi*t);

plot(ti,xi,xaxis=("time [s]"),title = "Time Domain",label=false)
```

0.1.7 Function - Function to Array

[3]:
```
# convert a function into an array

function functionToArrayi(T0, Δt)

    i = 0       #step between reads
    count = 0   #array sentinel
    size = 1001    #inclusive

    a = zeros(size)

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

[3]:
```
functionToArrayi (generic function with 1 method)
```

[4]:
```
bi = functionToArrayi(0.004,0.000004); # the funtion of t is stored in array␣
 ↪form, which will be parsed to fft()
```

[5]:
```
using FFTW

Xi = abs.(fft(bi));

Ni = length(ti);
Δf = 1/(Ni*Δt);

#create array of freq values stored in f_axis.
if mod(Ni,2)==0     # case N even
    f_axis = (-Ni/2:Ni/2-1)*Δf*(1/1000);
else    # case N odd
    f_axis = (-(Ni-1)/2 : (Ni-1)/2)*Δf*(1/1000);
end

plot(f_axis,fftshift(Xi),xaxis=("frequency [kHz]"),title = "Frequency␣
 ↪Domain",label=false)
```

0.1.8 (ii) Plot the carrier sinusoid

[6]:
```
fci = 20000 #Hz

xii(t) = cos(2*pi*fci*t);

plot(ti,xii,xaxis=("time [s]"),title = "Time Domain",label=false)
```

0.1.9 Function - Function to Array

[7]:
```
# convert a function into an array

function functionToArrayii(T0, Δt)

    i = 0       #step between reads
    count = 0   #array sentinel
    size = 1001    #inclusive

    a = zeros(size)

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

[7]:
```
functionToArrayii (generic function with 1 method)
```

[8]:
```
bii = functionToArrayii(0.004,0.000004); # the funtion of t is stored in array␣
 ↪form, which will be parsed to fft()
```

[9]:
```
Xii = abs.(fft(bii));

Ni = length(ti);
Δf = 1/(Ni*Δt);

#create array of freq values stored in f_axis.
if mod(Ni,2)==0    # case N even
    f_axis = (-Ni/2:Ni/2-1)*Δf*(1/1000);
else   # case N odd
    f_axis = (-(Ni-1)/2 : (Ni-1)/2)*Δf*(1/1000);
end

plot(f_axis,fftshift(Xii),xaxis=("frequency [kHz]"),title = "Frequency␣
 ↪Domain",label=false)
```

0.1.10 (iii) Plot the amplitude modulated carrier wave

[10]:
```
xiii(t) = xi(t)*xii(t);

plot(ti,xiii,xaxis=("time [s]"),title = "Time Domain",label=false)
```

0.1.11 Function - Function to Array

[11]:
```
# convert a function into an array

function functionToArrayiii(T0, Δt)

    i = 0       #step between reads
    count = 0   #array sentinel
    size = 1001     #inclusive

    a = zeros(size)

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

[11]:
```
functionToArrayiii (generic function with 1 method)
```

[12]:
```
biii = functionToArrayiii(0.004,0.000004); # the funtion of t is stored in␣
 ↪array form, which will be parsed to fft()
```

[13]:
```
# Applying zero padding
Nii = length(biii)
mi = zeros(16*Nii) # Make array 16x longer.
mi[1:Nii] = biii; # Copy x into first N samples. The rest contains zeros.

Xiii = abs.(fft(mi));

Ni = length(mi);
Δf = 1/(Ni*Δt);

#create array of freq values stored in f_axis.
if mod(Ni,2)==0     # case N even
    f_axis = (-Ni/2:Ni/2-1)*Δf*(1/1000);
else    # case N odd
    f_axis = (-(Ni-1)/2 : (Ni-1)/2)*Δf*(1/1000);
end
```

```
#print(length(f_axis))
plot(f_axis,fftshift(Xiii),xaxis=("frequency [kHz]"),title = "Frequency␣
↪Domain",label=false)
```

- lower sideband = +19.20 kHz,-19.20 kHz
- upper sideband = +21.10 kHz,-21.10 kHz

0.1.12 (iv) Using a 1kHz square wave modulating waveform

0.1.13 Plot the modulating waveform

[14]:
```
square_wavei(t) = ( (sin.(2*pi*1000*t) .> 0) .- 0.5 )*2; # square wave -1,+1

plot(square_wavei,-0.0015,0.0025,xaxis=("time [s]"),title = "Time␣
↪Domain",label=false)
```

0.1.14 Function - Function to Array

[15]:
```
# convert a function into an array

function functionToArrayiv(T0, Δt)

    i = 0       #step between reads
    count = 0   #array sentinel
    size = 1001     #inclusive

    a = zeros(size)

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

[15]:
```
functionToArrayiv (generic function with 1 method)
```

[16]:
```
biv = functionToArrayiv(0.004,0.000004); # the funtion of t is stored in array␣
↪form, which will be parsed to fft()
```

[17]:
```
Xiv = abs.(fft(biv));

Ni = length(ti);
Δf = 1/(Ni*Δt);
```

```
#create array of freq values stored in f_axis.
if mod(Ni,2)==0    # case N even
    f_axis = (-Ni/2:Ni/2-1)*Δf*(1/1000);
else   # case N odd
    f_axis = (-(Ni-1)/2 : (Ni-1)/2)*Δf*(1/1000);
end

plot(f_axis,fftshift(Xiv),xaxis=("frequency [kHz]"),title = "Frequency␣
 ↪Domain",label=false)
```

0.1.15 Plot the amplitude modulated carrier wave

[18]:
```
 i(t) = xi(t)*square_wavei(t);

plot(ti, i,xaxis=("time [s]"),title = "Time Domain",label=false)
```

0.1.16 Function - Function to Array

[19]:
```
# convert a function into an array

function functionToArrayv(T0, Δt)

    i = 0       #step between reads
    count = 0   #array sentinel
    size = 1001    #inclusive

    a = zeros(size)

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

[19]:
```
functionToArrayv (generic function with 1 method)
```

[20]:
```
b i = functionToArrayv(0.004,0.000004); # the funtion of t is stored in array␣
 ↪form, which will be parsed to fft()
```

[21]:
```
X i = abs.(fft(b i));

Ni = length(ti);
Δf = 1/(Ni*Δt);
```

```
#create array of freq values stored in f_axis.
if mod(Ni,2)==0    # case N even
    f_axis = (-Ni/2:Ni/2-1)*Δf*(1/1000);
else   # case N odd
    f_axis = (-(Ni-1)/2 : (Ni-1)/2)*Δf*(1/1000);
end

plot(f_axis,fftshift(X i),xaxis=("frequency [kHz]"),title = "Frequency␣
↪Domain",label=false)
```

0.1.17 Julia Exercise 6.1b – DSB-SC Demodulation

0.1.18 (i) Plot the DSB-SC Demodulation

[22]:
```
ii(t) = xiii(t)*xii(t);

plot(ti, ii,xaxis=("time [s]"),title = "Time Domain",label=false)
```

0.1.19 Function - Function to Array

[23]:
```
# convert a function into an array

function functionToArrayvi(T0, Δt)

    i = 0       #step between reads
    count = 0   #array sentinel
    size = 1001     #inclusive

    a = zeros(size)

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

[23]:
```
functionToArrayvi (generic function with 1 method)
```

[24]:
```
b ii = functionToArrayvi(0.004,0.000004); # the funtion of t is stored in array␣
↪form, which will be parsed to fft()
```

[25]:
```
X ii = abs.(fft(b ii));

Ni = length(ti);
Δf = 1/(Ni*Δt);
```

```
#create array of freq values stored in f_axis.
if mod(Ni,2)==0     # case N even
    f_axis = (-Ni/2:Ni/2-1)*Δf*(1/1000);
else    # case N odd
    f_axis = (-(Ni-1)/2 : (Ni-1)/2)*Δf*(1/1000);
end

plot(f_axis,fftshift(X ii),xaxis=("frequency [kHz]"),title = "Frequency␣
↪Domain",label=false)
```

0.1.20 (ii) Implement an ideal LPF (H) in the frequency domain.

0.1.21 Plot a LPF with a cut-off at 5kHz

[26]:
```
Δ  = 2*pi/(Ni*Δt)    # Sample spacing in freq domain in rad/s

  = 0:Δ :(Ni-1)*Δ
B = 5000 # filter bandwidth in Hz

#create array of freq values stored in f_axis.
if mod(Ni,2)==0     # case N even
    f_axis2 = (-Ni/2:Ni/2-1)*Δf*(1/1000);
else    # case N odd
    f_axis2 = (-(Ni-1)/2 : (Ni-1)/2)*Δf*(1/1000);
end

rect(t) = (abs.(t).<=0.5)*1.0

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

plot(f_axis2,fftshift(H),xaxis=("frequency [kHz]"),title = "Low-Pass␣
↪Filter",label=false)
```

0.1.22 Apply the LPF

[27]:
```
Yi = X ii.*H

plot(f_axis2,fftshift(Yi),xaxis=("frequency [kHz]"),title = "Demodulated Output␣
↪- Frequency Domain",label=false)
```

[28]:
```
yi = real(ifft(Yi));

plot(ti,-yi,xaxis=("time [s]"),title = "Demodulated Output - Time␣
↪Domain",label=false)
```

Does the final output agree with the theory? Do you get out ½f(t)?

Yes, the final output agrees with the theory. The modulating waveform had a maximum amplitude of 1, while the demodulation output waveform has a maximum amplitude of 0.5.

0.1.23 Julia Exercise 6.1c – Effect of phase error in DSC-SC demodulation

[29]:
```
function phaseErrori( )

    rad =  *(pi/180)

    plot(ti,cos(rad)*-yi,xaxis=("time [s]"),title = "Demodulated Output - Time␣
 ↪Domain,   = $( ) degrees",label=false)
    ylims!((-0.5,0.5));
end
```

[29]:
```
phaseErrori (generic function with 1 method)
```

0.1.24 (i) Phase error of 30 degrees

[30]:
```
phaseErrori(30)
```

0.1.25 (ii) Phase error of 60 degrees

[31]:
```
phaseErrori(60)
```

0.1.26 (iii) Phase error of 85 degrees

[32]:
```
phaseErrori(85)
```

0.1.27 (iv) Phase error of 90 degrees

[33]:
```
phaseErrori(90)
```

0.1.28 Julia Exercise 6.1d – DSB-SC AM with Gaussian noise

0.1.29 Simulate a noise waveform

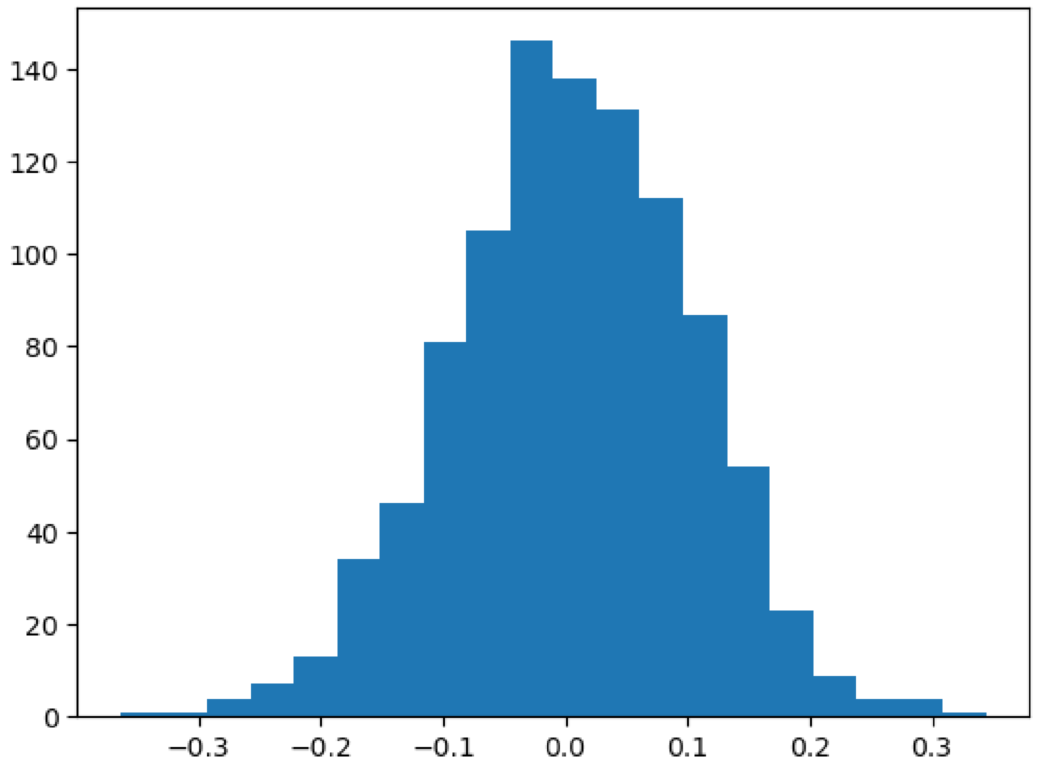
[34]:
```
using PyPlot
```

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

[35]:
```
 i = 0.1;
Nii = 1001;
nbins = 20

noise =  i*randn(Nii);
```

```
PyPlot.hist(noise,nbins);
```



[36]:
```
Ni = abs.(fft(noise));

if mod(Nii,2)==0     # case N even
    t_axis = (-Nii/2:Nii/2-1)*Δt;
else    # case N odd
    t_axis = (-(Nii-1)/2 : (Nii-1)/2)*Δt;
end

plot(t_axis,Ni,xaxis=("time [s]"),title = "Noise Waveform",label=false)
```

0.1.30 Plot noisy AM signal

[37]:
```
nxi = Ni.+bi; # modulating signal in array bi
plot(t_axis,nxi,xaxis=("time [s]"),title = "Noise Waveform",label=false)
```

[38]:
```
Vo = abs.(fft(nxi.*bii)).*H # carrier signal in array bii

plot(t_axis,fftshift(Vo),xaxis=("time [s]"),title = "Demodulated Output - Time␣
↪Domain",label=false)
```

What effect does a phase shift error have on the output SNR?

No effect to the output SNR. The signal and the noise are both multiplied by the cos , causing its effects to cancel out while calculating the signal to noise ratio.

How would you calculate the SNR at the output of the demodulator?

I would demodulate the noise signal and the DSB-SC signal separately, pass them through a low-pass filter, and finally calculate the ratio of the signal to the noise.

0.2 Julia Exercise 6.2 – DSB-LC AM

0.2.1 Julia Exercise 6.2a – DSB-LC AM Modulation

0.2.2 Simulate double sideband large carrier amplitude modulation (DSB-LC) with the following parameters:

0.2.3 Modulating waveform: f(t) = k cos(2 fmt) where fm=1 kHz and k is a constant (k<A).

0.2.4 Carrier wave oscillator: cos(2 fct) where fc=20 kHz.

0.2.5 Modulated carrier wave signal: (t) = Acos(2 fct) + f(t) cos(2 fct) where A is a constant. Let A=1 for these simulations. Initially, let k=0.5

0.2.6 (i) Plot the modulating waveform

[39]:
```
k = 0.5;

vi(t) = k*cos(2*pi*fmi*t);

plot(ti,vi,xaxis=("time [s]"),title = "Time Domain",label=false)
```

0.2.7 Function - Function to Array

[40]:
```
# convert a function into an array

function functionToArrayvii(T0, Δt)

    i = 0       #step between reads
    count = 0   #array sentinel
    size = 1001    #inclusive

    a = zeros(size)

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

```
end
```

[40]:
```
functionToArrayvii (generic function with 1 method)
```

[41]:
```
ci = functionToArrayvii(0.004,0.000004); # the funtion of t is stored in array␣
↪form, which will be parsed to fft()
```

[42]:
```
Vi = abs.(fft(ci));

Ni = length(ti);
Δf = 1/(Ni*Δt);

#create array of freq values stored in f_axis.
if mod(Ni,2)==0    # case N even
    f_axis = (-Ni/2:Ni/2-1)*Δf*(1/1000);
else    # case N odd
    f_axis = (-(Ni-1)/2 : (Ni-1)/2)*Δf*(1/1000);
end

plot(f_axis,fftshift(Vi),xaxis=("frequency [kHz]"),title = "Frequency␣
↪Domain",label=false)
```

0.2.8 Plot the carrier sinusoid

[43]:
```
vii(t) = cos(2*pi*fci*t);

plot(ti,vii,xaxis=("time [s]"),title = "Time Domain",label=false)
```

0.2.9 Function - Function to Array

[44]:
```
# convert a function into an array

function functionToArrayviii(T0, Δt)

    i = 0       #step between reads
    count = 0   #array sentinel
    size = 1001     #inclusive

    a = zeros(size)

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

```
end
```

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

[45]:
```
cii = functionToArrayviii(0.004,0.000004); # the funtion of t is stored in␣
↪array form, which will be parsed to fft()
```

[46]:
```
Vii = abs.(fft(cii));

Ni = length(ti);
Δf = 1/(Ni*Δt);

#create array of freq values stored in f_axis.
if mod(Ni,2)==0     # case N even
    f_axis = (-Ni/2:Ni/2-1)*Δf*(1/1000);
else    # case N odd
    f_axis = (-(Ni-1)/2 : (Ni-1)/2)*Δf*(1/1000);
end

plot(f_axis,fftshift(Vii),xaxis=("frequency [kHz]"),title = "Frequency␣
↪Domain",label=false)
```

0.2.10 Plot the amplitude modulated carrier wave

[47]:
```
A = 1;

viii(t) = vii(t)*(vi(t)+A)

plot(ti,viii,xaxis=("time [s]"),title = "Time Domain",label=false)
```

0.2.11 Function - Function to Array

[48]:
```
# convert a function into an array

function functionToArrayix(T0, Δt)

    i = 0       #step between reads
    count = 0   #array sentinel
    size = 1001    #inclusive

    a = zeros(size)

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

[48]:
```
functionToArrayix (generic function with 1 method)
```

[49]:
```
ciii = functionToArrayix(0.004,0.000004); # the funtion of t is stored in array␣
 ↪form, which will be parsed to fft()
```

[50]:
```
# Applying zero padding
Niii = length(ciii)
mii = zeros(16*Niii) # Make array 16x longer.
mii[1:Niii] = ciii; # Copy x into first N samples. The rest contains zeros.

Viii = abs.(fft(mii));

Ni = length(mii);
Δf = 1/(Ni*Δt);

#create array of freq values stored in f_axis.
if mod(Ni,2)==0    # case N even
    f_axis = (-Ni/2:Ni/2-1)*Δf*(1/1000);
else   # case N odd
    f_axis = (-(Ni-1)/2 : (Ni-1)/2)*Δf*(1/1000);
end

plot(f_axis,fftshift(Viii),xaxis=("frequency [kHz]"),title = "Frequency␣
 ↪Domain",label=false)
```

- lower sideband = +19.20 kHz,-19.20 kHz
- upper sideband = +21.10 kHz,-21.10 kHz
- carrier = +20.14 kHz,-20.14 kHz

What is the modulation index for this case?

Envelope maxima

A(1+m) = 1.5

A = 1.5/(1+m)

Envelope minima

A(1-m) = 0.5

A = 0.5/(1-m)

Therefore:

1.5/(1+m) = 0.5/(1-m)

1.5 - 1.5m = 0.5 + 0.5m

2m = 1

m = 0.5

0.2.12 Julia Exercise 6.2b – DSB-LC AM Demodulation

0.2.13 Full-wave rectify the AM signal

[51]:
```
plot(ti,abs.(ciii),xaxis=("time [s]"),title = "Time Domain",label=false)
```

0.2.14 Plot a LPF with a cut-off at 5kHz

[52]:
```
plot(f_axis2,fftshift(H),xaxis=("frequency [kHz]"),title = "Low-Pass␣
 ↪Filter",label=false)
```

0.2.15 Plot a BPF with a passband of 20Hz to 5kHz

[53]:
```
H1b = rect(( .+2* *12500)/(1.5*4* *B)) + rect( (( .+2* *12500) .- 2* /Δt)/(1.
 ↪5*4* *B) ) # negative frequencies
H2b = rect(( .-2* *12500)/(1.5*4* *B)) + rect( (( .-2* *12500) .- 2* /Δt)/(1.
 ↪5*4* *B) ) # positive frequencies
Hb = H1b + H2b;

plot(f_axis2,fftshift(Hb),xaxis=("frequency [kHz]"),title = "Band-Pass␣
 ↪Filter",label=false)
```

0.2.16 Plot the output of the low-pass filter

[54]:
```
output_of_lpfilter = real( ifft( fft(abs.(ciii)).*H ));

plot(ti,-output_of_lpfilter,xaxis=("time [s]"),title = "Low-Pass Filter␣
 ↪Output",label=false)
```

0.2.17 Plot the output of the band-pass filter

[55]:
```
output_of_bpfilter = real( ifft( fft(abs.(ciii)).*Hb ));

plot(ti,-output_of_bpfilter,xaxis=("time [s]"),title = "Band-Pass Filter␣
 ↪Output",label=false)
```

0.2.18 Julia Exercise 6.2c – DSB-LC AM modulation index

[56]:
```
function modulationIndex(m)

    Ai = 1;
    viv(t) = m*Ai*cos(2*pi*fmi*t);
    vv(t) = cos(2*pi*fci*t);

    vvi(t) = vv(t)*(viv(t)+Ai)

    # convert a function into an array

    i = 0       #step between reads
    count = 0   #array sentinel
    size = 1001    #inclusive

    a = zeros(size)

    while i < 0.004
        a[count+1] = vvi(i)
        i += 0.000004;
        count += 1;
    end

    output_of_lpfilter = real( ifft( fft(abs.(a)).*H ));

    plot(ti,-output_of_lpfilter,xaxis=("time [s]"),title = "Low-Pass Filter␣
↪Output, m = $(m)",label=false)

end
```

[56]: `modulationIndex (generic function with 1 method)`

0.2.19 Fine modulation

[57]:
```
modulationIndex(0.5)
```

0.2.20 Critically modulated

[58]:
```
modulationIndex(1)
```

0.2.21 Over modulated

[59]:
```
modulationIndex(2)
```

In the over modulated case, f(t) cannot be recovered from the envelope.

This is because the envelope dip A(1-m) is at A(1-2), which dives below a 0 value, -A. In DSB-LC AM, the envelope should always be above 0; therefore, m should always be less than 1.

0.3 Julia Exercise 6.3 – Quadrature Multiplexing

0.3.1 Plot all inputs signals

[60]:
```
fm1 = 200 #Hz

startTime1 = -0.005;
stopTime1 = 0.010;
Δt1 = (stopTime1-startTime1)/1000;

tii = startTime1:Δt1:stopTime1;

x1(t) = cos(2*pi*fm1*t);

plot(tii,x1,xaxis=("time [s]"),title = "Input x1(t)",label=false)
```

[61]:
```
fm2 = 1000 #Hz

x2(t) = cos(2*pi*fm2*t);

plot(tii,x2,xaxis=("time [s]"),title = "Input x2(t)",label=false)
```

0.3.2 Plot all modulated signals prior to addition

[62]:
```
fc = 10000 #Hz

 1(t) = x1(t)*cos(2*pi*fc*t)

plot(tii, 1,xaxis=("time [s]"),title = "Signal  1(t)",label=false)
```

[63]:
```
 2(t) = x2(t)*sin(2*pi*fc*t)

plot(tii, 2,xaxis=("time [s]"),title = "Signal  2(t)",label=false)
```

0.3.3 Plot the quadrature multiplexed carrier wave

0.3.4 Time Domain

[64]:
```
 (t) =  1(t) +  2(t);

plot(tii, ,xaxis=("time [s]"),title = "Quadrature Multiplexed Carrier␣
 ↪Wave",label=false)
```

0.3.5 Frequency Domain

0.3.6 Function - Function to Array

[65]:
```
# convert a function into an array

function functionToArrayx(T0, Δt)

    i = 0        #step between reads
    count = 0    #array sentinel
    size = 1001     #inclusive

    a = zeros(size)

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

[65]:
```
functionToArrayx (generic function with 1 method)
```

[66]:
```
arr = functionToArrayx(0.015,0.000015); # the funtion of t is stored in array␣
 ↪form, which will be parsed to fft()
```

[67]:
```
# Applying zero padding
Niv = length( arr)
miii = zeros(16*Niv) # Make array 16x longer.
miii[1:Niv] =  arr; # Copy x into first N samples. The rest contains zeros.

  = abs.(fft(miii));

Ni = length(miii);
Δf = 1/(Ni*Δt);

#create array of freq values stored in f_axis.
if mod(Ni,2)==0     # case N even
    f_axis = (-Ni/2:Ni/2-1)*Δf*(1/1000);
else    # case N odd
    f_axis = (-(Ni-1)/2 : (Ni-1)/2)*Δf*(1/1000);
end

plot(f_axis,fftshift( ),xaxis=("frequency [kHz]"),title = "Frequency␣
 ↪Domain",label=false)
```

0.3.7 Plot all demodulated signals prior to filtering

0.3.8 Signal y1(t)

[68]:
```
y1(t) =  (t)*cos(2*pi*fc*t)

plot(tii,y1,xaxis=("time [s]"),title = "Time Domain - y1(t)",label=false)
```

0.3.9 Function - Function to Array

[69]:
```
# convert a function into an array

function functionToArrayxi(T0, Δt)

    i = 0        #step between reads
    count = 0    #array sentinel
    size = 1001     #inclusive

    a = zeros(size)

    while i < T0
        a[count+1] = y1(i)
        i += Δt;
        count += 1;
    end
    return a
end
```

[69]:
```
functionToArrayxi (generic function with 1 method)
```

[70]:
```
y1arr = functionToArrayxi(0.015,0.000015); # the funtion of t is stored in␣
 ↪array form, which will be parsed to fft()
plot(y1arr,xaxis=("frequency [kHz]"),title = "Frequency Domain -␣
 ↪y1(t)",label=false)
```

[71]:
```
# Applying zero padding

Y1 = abs.(fft(y1arr));

Ni = length(y1arr);
Δf = 1/(Ni*Δt);

#create array of freq values stored in f_axis.
if mod(Ni,2)==0     # case N even
    f_axis = (-Ni/2:Ni/2-1)*Δf*(1/1000);
else    # case N odd
    f_axis = (-(Ni-1)/2 : (Ni-1)/2)*Δf*(1/1000);
```

```
end

plot(f_axis,fftshift(Y1),xaxis=("frequency [kHz]"),title = "Frequency Domain -␣
↪y1(t)",label=false)
```

0.3.10 Signal y2(t)

[72]:
```
y2(t) =  (t)*sin(2*pi*fc*t)

plot(tii,y2,xaxis=("time [s]"),title = "Time Domain - y2(t)",label=false)
```

0.3.11 Function - Function to Array

[73]:
```
# convert a function into an array

function functionToArrayxii(T0, Δt)

    i = 0       #step between reads
    count = 0   #array sentinel
    size = 1001    #inclusive

    a = zeros(size)

    while i < T0
        a[count+1] = y2(i)
        i += Δt;
        count += 1;
    end
    return a
end
```

[73]:
```
functionToArrayxii (generic function with 1 method)
```

[74]:
```
y2arr = functionToArrayxii(0.015,0.000015); # the funtion of t is stored in␣
↪array form, which will be parsed to fft()
```

[75]:
```
# Applying zero padding

Y2 = abs.(fft(y2arr));

Ni = length(y2arr);
Δf = 1/(Ni*Δt);

#create array of freq values stored in f_axis.
if mod(Ni,2)==0    # case N even
    f_axis = (-Ni/2:Ni/2-1)*Δf*(1/1000);
```

```
else   # case N odd
    f_axis = (-(Ni-1)/2 : (Ni-1)/2)*Δf*(1/1000);
end

plot(f_axis,fftshift(Y2),xaxis=("frequency [kHz]"),title = "Frequency Domain -␣
 ↪y2(t)",label=false)
```

0.3.12 Plot the Low-Pass Filter

[76]:
```
Δ  = 2*pi/(Ni*Δt)   # Sample spacing in freq domain in rad/s

  = 0:Δ :(Ni-1)*Δ
B = 1500 # filter bandwidth in Hz

#create array of freq values stored in f_axis.
if mod(Ni,2)==0    # case N even
    f_axis2 = (-Ni/2:Ni/2-1)*Δf*(1/1000);
else   # case N odd
    f_axis2 = (-(Ni-1)/2 : (Ni-1)/2)*Δf*(1/1000);
end

rect(t) = (abs.(t).<=0.5)*1.0

H1 = rect( /(4* *B)) + rect( (  .- 2* /Δt)/(4* *B) );

plot(f_axis2,fftshift(H1),xaxis=("frequency [kHz]"),title = "Low-Pass␣
 ↪Filter",label=false)
xlims!((-5,5))
```

0.3.13 Plot the Band-Pass Filter

[77]:
```
Bbpf = 1000 # filter bandwidth in Hz

Hbpf1 = rect(( .+2* *4000)/(2* *Bbpf)) + rect( (( .+2* *4000) .- 2* /Δt)/
 ↪(2* *Bbpf) ) # negative frequencies
Hbpf2 = rect(( .-2* *4000)/(2* *Bbpf)) + rect( (( .-2* *4000) .- 2* /Δt)/
 ↪(2* *Bbpf) ) # positive frequencies
Hbpf = Hbpf1 + Hbpf2;

plot(f_axis2,fftshift(Hbpf),xaxis=("frequency [kHz]"),title = "Band-Pass␣
 ↪Filter",label=false)
xlims!((-5,5))
```

0.3.14 Plot all final outputs

[78]:
```
lpfilter_output_y1 = real( ifft( (fft(y1arr)).*H1 )); # low-pass filter used

plot(tii,lpfilter_output_y1,xaxis=("time [s]"),title = "Low-Pass Filter Output␣
↪- e1(t)",label=false)
```

[79]:
```
lpfilter_output_y2 = real( ifft( fft(y2arr).*Hbpf )); # band-pass filter used

plot(tii,lpfilter_output_y2,xaxis=("time [s]"),title = "Band-Pass Filter Output␣
↪- e2(t)",label=false)
```

What is the minimum sample rate that you can use for this system? How did you determine it?

fs > 2B, given Bmax = 1kHz

fs > 2kHz i.e. 2000 samples every second

If x1(t) has bandwidth B1 and x2(t) has bandwidth B2, what is the bandwidth of the transmitted waveform (t)?

B2 (1kHz) since it is greater than B1 (200 Hz)

What happens if the sin and cos oscillators are not perfectly in quadrature?

Upper Arm experiences additive interference from the second signal: 0.5x1(t) + 0.5x2(t)sin( )

Lower Arm experiences additive interference from the first signal: 0.5x2(t) + 0.5x1(t)sin( )

0.3.15 Case of Imperfect quadrature - sin and cos oscillators

[80]:
```
function imperfectQuadrature( )

    # both modulator and demodulator oscillators are in phase, just the arms␣
↪phase varies
    v1(t) = 0.5*(sin( *pi/180)*x2(t)) + 0.5*x1(t); # after calculation
    v2(t) = 0.5*(sin( *pi/180)*x1(t)) + 0.5*x2(t); # after calculation

    plot1 = plot(tii,v1,title = "e1(t)",label=false)
    plot2 = plot(tii,v2,title = "e2(t)",label=false)

    plot(plot1,plot2,xaxis=("time [s]"),label=false)
    ylims!((-0.5,0.5))
end
```

[80]:
```
imperfectQuadrature (generic function with 1 method)
```

0.3.16 Case = 0 deg

[81]:
```
imperfectQuadrature(0)
```

0.3.17 Case = 60 deg

[82]:
```
imperfectQuadrature(60)
```

0.3.18 Case = 90 deg

[83]:
```
imperfectQuadrature(90)
```

[ ]:
```

```