Lab2_notebook_sample

February 18, 2022

1 Lab 2 Sample Notebook

[ ]:
```
%pylab inline
#%pylab notebook
#%matplotlib qt
import sk_dsp_comm.sigsys as ss
import scipy.signal as signal
import ipywidgets as widgets
from ipywidgets import interact, interactive, fixed, interact_manual
from IPython.display import Audio, display
from IPython.display import Image, SVG
```

```
Populating the interactive namespace from numpy and matplotlib
```

[ ]:
```
pylab.rcParams['savefig.dpi'] = 100 # default 72
#pylab.rcParams['figure.figsize'] = (6.0, 4.0) # default (6,4)
#%config InlineBackend.figure_formats=['png'] # default for inline viewing
%config InlineBackend.figure_formats=['svg'] # SVG inline viewing
#%config InlineBackend.figure_formats=['pdf'] # render pdf figs for LaTeX
#<div style="page-break-after: always;"></div> #page breaks after in Typora
```

1.1 Plotting Line Spectra

[ ]:
```
def line_spectra_dBm(fn,Xn,floor_dBm = -60,R0 = 50):
    """
    Plot the one-sided line spectra from Fourier coefficients in dBm.

    Inputs
    ------
    fn = harmonic frequencies correspinding to Xn's
    Xn = pulse train FS coefficients
    floor_dBm = spectrum floor in dBm
    R0 = impedance (default 50 ohms)

    Returns
    -------
    Sx_dBm = One-sided bandpass lines as dBm levels
```

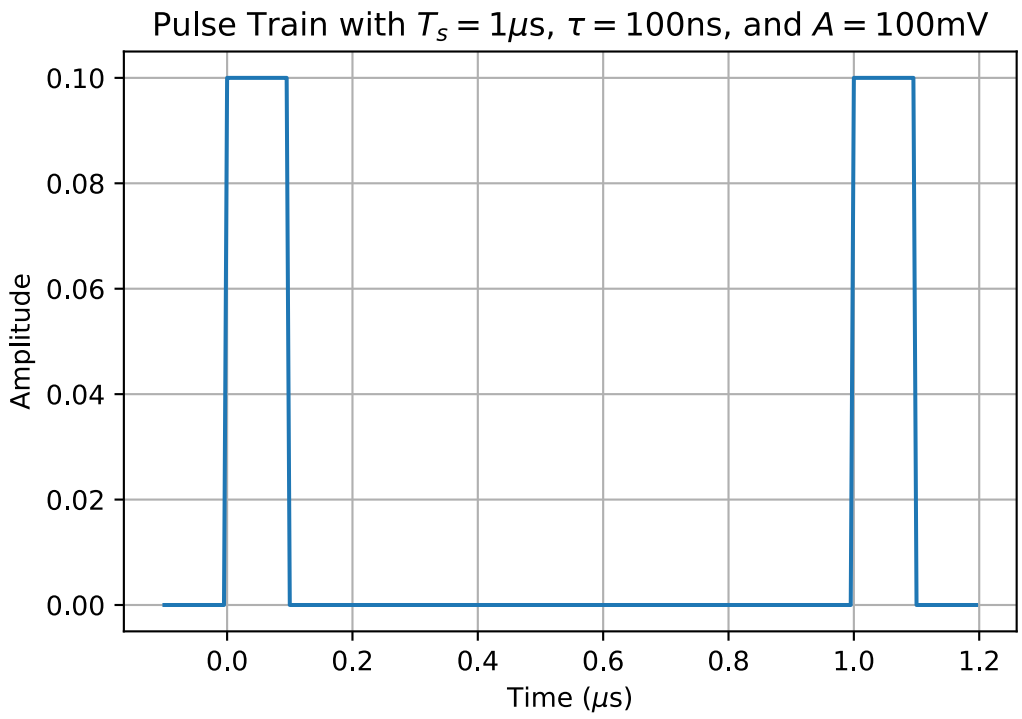
```
    Mark Wickert February 2019
    """
    Xn_dBm = abs(Xn*sqrt(1000/(2*R0)))
    Xn_dBm[0] = abs(Xn[0])*sqrt(1000/(R0))

    ss.line_spectra(fn,Xn_dBm,mode='magdB',sides=1,floor_dB=floor_dBm)
    ylabel(r'Power (dBm)')
    xlabel(r'Frequency (Hz)')
    title(r'Line Spectra PSD')
    Sx_dBm = 20*log10(2*Xn_dBm)
    Sx_dBm[0] -= 6.02
    return Sx_dBm
```

1.1.1 Consider a Pulse train at 1 MHz

Set the pulse width to 100 ns and the peak amplitude of the pulse train into a 50 ohm load of 100 mV.

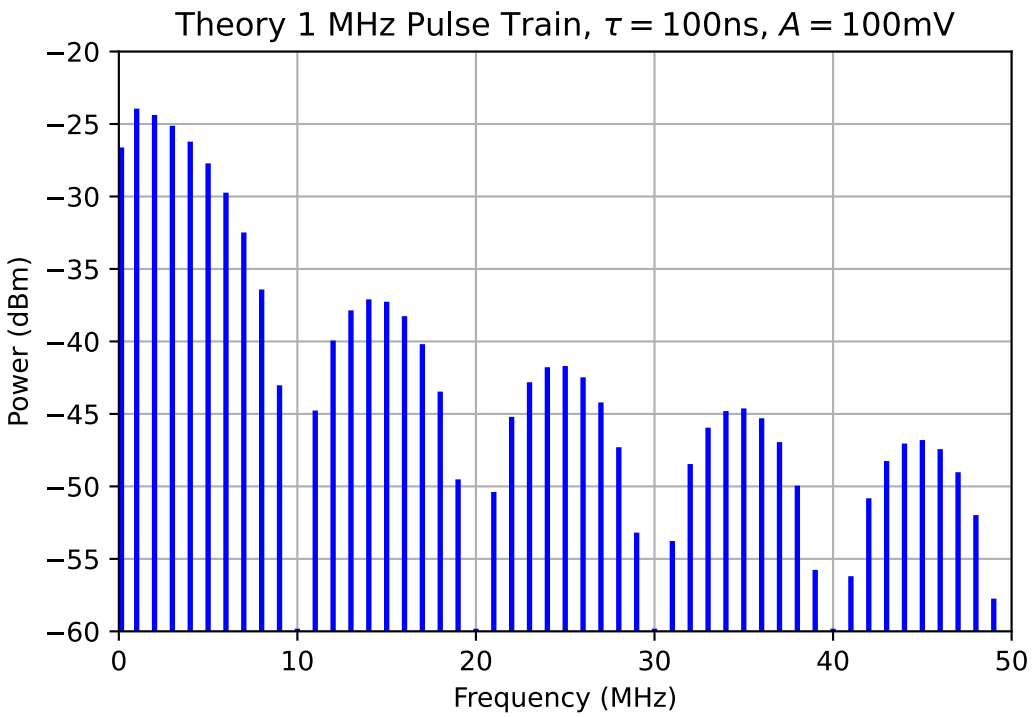
Time Domain

[ ]:
```
# Plot 1.2 periods
Ts = 1e-6
tau = 100e-9
A = 0.1
t = arange(-.1*Ts,1.2*Ts,tau/20)
x_PT = A*ss.rect(t-tau/2,tau) + A*ss.rect(t-Ts-tau/2,tau)
plot(t*1e6,x_PT)
title(r'Pulse Train with $T_s=1\mu$s, $\tau=100$ns, and $A = 100$mV')
ylabel(r'Amplitude')
xlabel(r'Time ($\mu$s)')
#axis('off')
grid();
#savefig('pt.pdf')
```



Frequency Domain: Power Spectrum in dBm

[ ]:
```
n = arange(0,50)
Ts = 1e-6
tau = 100e-9
A = 0.1
Xn = A*tau/Ts*sinc(n*tau/Ts)*exp(-1j*2*pi*n*tau/2/Ts)
Xn_dBm = line_spectra_dBm(n/Ts/1e6,Xn,floor_dBm = -60)
title(r'Theory 1 MHz Pulse Train, $\tau = 100$ns, $A = 100$mV')
xlabel(r'Frequency (MHz)');
xlim([0,50]);
```



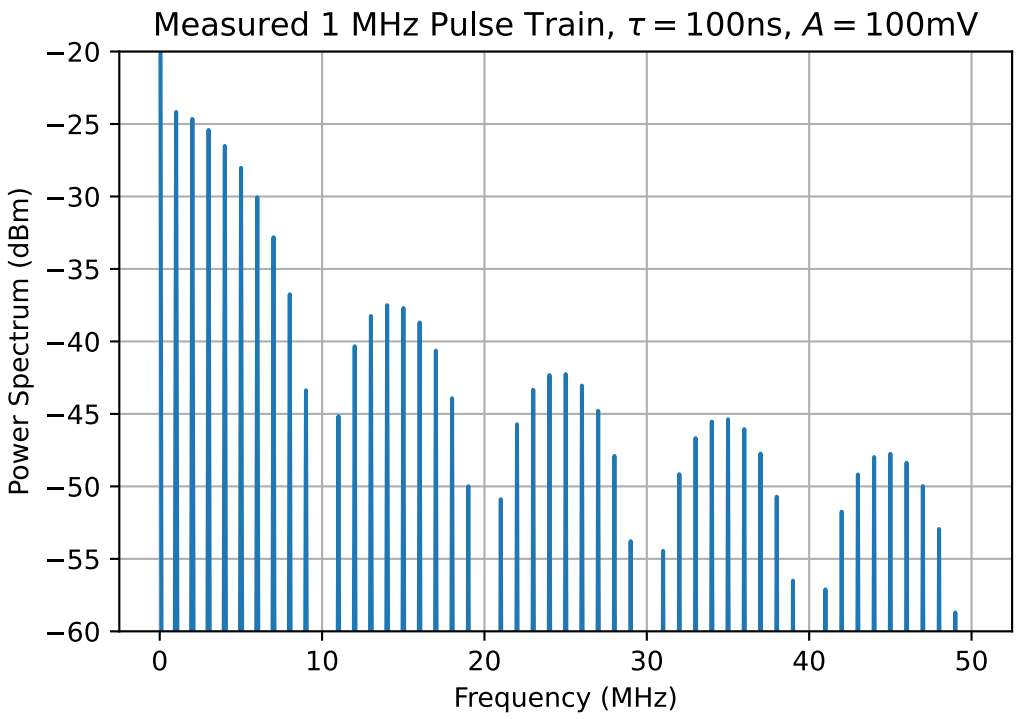
[ ]:
```
20*log10(abs(Xn[1])) + 10*log10(2/50) + 30
```

[ ]:
```
-24.12275025888225
```

1.1.2 Import 1 MHz Pulse Train Data and Plot the Measured Power Spectrum

[ ]:
```
# Skip the first 32 rows, then skip the last row that contains 'END'
f_SA, Sx_1M_10duty_100mV = loadtxt('PULSE_1M_10duty_100mV.
↪csv',delimiter=',',skiprows=32,
                usecols=(0,1),comments='END',unpack=True)
```

[ ]:
```
# Note the spectral line at DC is from the Field Fox itself
# A coupling capacitor prevents DC from entering the analyzer.
plot(f_SA/1e6,Sx_1M_10duty_100mV)

xlabel(r'Frequency (MHz)')
ylabel(r'Power Spectrum (dBm)')
title(r'Measured 1 MHz Pulse Train, $\tau = 100$ns, $A = 100$mV')
ylim([-60,-20])
grid()
```



1.1.3 Compare Theory and Measured Spectral Peaks

Theoretical Peaks

[ ]:
```
for k, Xn_dBm_k in enumerate(Xn_dBm):
    if Xn_dBm_k > -30: # Threshold = -30 dBm
        print('Peak at %6.4f MHz of height %6.4f dBm'% (n[k]/Ts/1e6,Xn_dBm[k]))
```

```
Peak at 0.0000 MHz of height -26.9891 dBm
Peak at 1.0000 MHz of height -24.1228 dBm
Peak at 2.0000 MHz of height -24.5586 dBm
Peak at 3.0000 MHz of height -25.3057 dBm
Peak at 4.0000 MHz of height -26.3995 dBm
Peak at 5.0000 MHz of height -27.9018 dBm
Peak at 6.0000 MHz of height -29.9213 dBm
```

Measured Peaks

[ ]:
```
peak_idx = signal.find_peaks(Sx_1M_10duty_100mV,height=(-30,)) # peak threshold
↪= -30 dBm
for k, k_idx in enumerate(peak_idx[0]):
    print('Peak at %6.4f MHz of height %6.4f dBm'% (f_SA[k_idx]/
↪1e6,Sx_1M_10duty_100mV[k_idx]))
```

```
Peak at 1.0098 MHz of height -24.1698 dBm
Peak at 2.0096 MHz of height -24.6501 dBm
```

```
Peak at 3.0094 MHz of height -25.4173 dBm
Peak at 4.0092 MHz of height -26.5120 dBm
Peak at 5.0090 MHz of height -28.0286 dBm
```

The agreement is excellent!

1.2 Cosinusoidal Signal Theory/Simulation

Consider the mathematical form of a coherent cosinusoidal signal:

$$x(t) = \sum_{m=-\infty}^{\infty} A\,\Pi\left(\frac{t - mT_s - N_{\text{cyc}}/(2f_0)}{N_{\text{cyc}}/f_0}\right) \sin\left[2\pi f_0(t - mT_s) + \theta\right],\ 0 < N_{\text{cyc}}/f_0 < T_s$$

and also the non-coherent form where the pulse train and sinusoid at f_0 can be free running with respect to each other:

$$x(t) = \sum_{m=-\infty}^{\infty} A\,\Pi\left(\frac{t - mT_s - N_{\text{cyc}}/(2f_0)}{N_{\text{cyc}}/f_0}\right) \cdot \sin(2\pi f_0 t + \theta),\ 0 < N_{\text{cyc}}/f_0 < T_s$$

1.2.1 Coherent Form

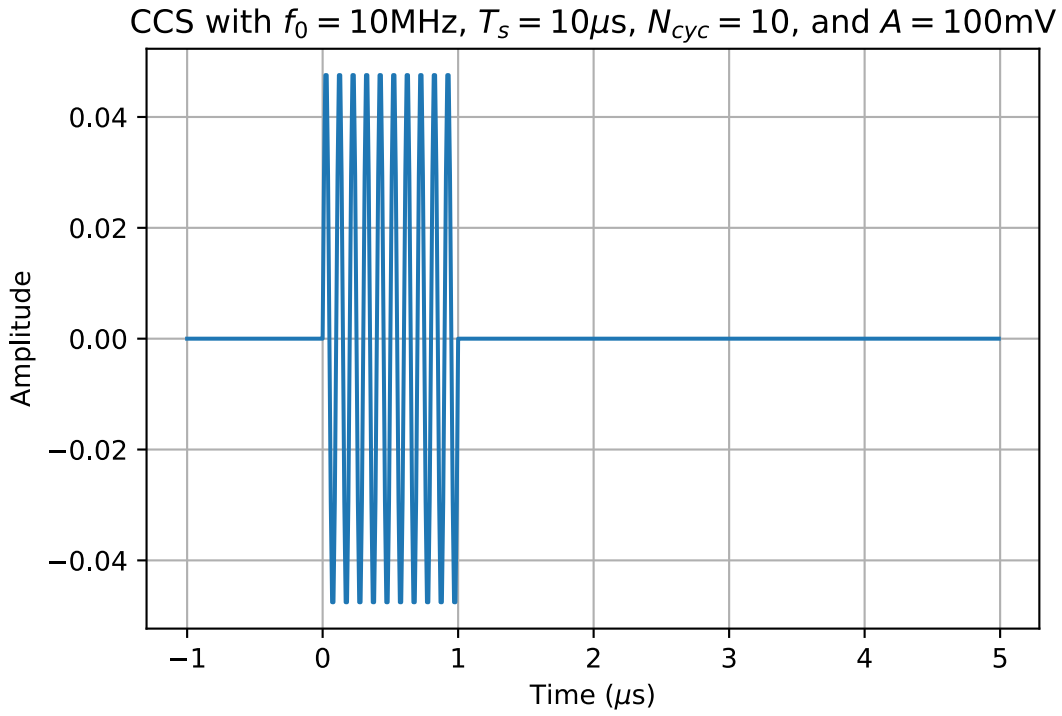
The Keysight 33600A produces the coherent form. The function below obtains the Fourier coefficients from $p(t)$, one full T_s period, then finds $P(f) = \mathcal{F}\{p(t)\}$.

[ ]:
```
def coherent_cosinusoid(f0,A,Ncyc,Ts,Nstop,Nstart=0,theta=0):
    """
    Coherent cosinusoid pulse train Xn Fourier coefficients

    Inputs
    ------
        f0 = carrier frequency in Hz
         A = peak amplitude (2*A = p-p)
      Ncyc = cycle per period
        Ts = pulse train period
     Nstop = the coefficients stop index (actually one less)
    Nstart = the coefficient start index (default is 0)
     theta = starting phase (default = 0)

    Returns
    -------
    Xn = Fourier coefficients over the specified coefficient range

    Note: Ncyc*1/f0 must be less than Ts

    Mark Wickert February 2019
    """
    Xn = zeros(Nstop-Nstart,dtype = complex128)
    for n in range(Nstart,Nstop):
        beta = Ncyc*(1 - n/(Ts*f0))
```

```
        alpha = Ncyc*(1 + n/(Ts*f0))
        Xn[n-Nstart] = A*Ncyc/(2*Ts*f0)*(sinc(beta)*exp(1j*(pi*beta + theta)) \
              - sinc(alpha)*exp(-1j*(pi*alpha + theta)))
    return Xn
```

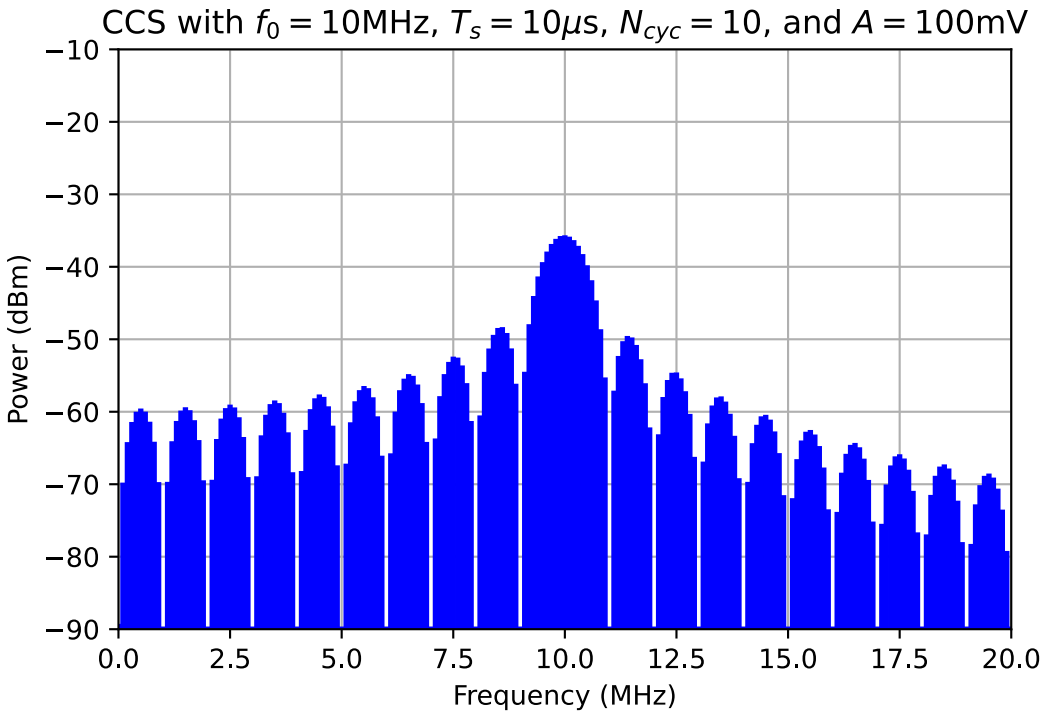
Time Domain

[ ]:
```
# Plot ~.5 periods
f0 = 10.0e6
A = 0.1/2
Ncyc = 10
Ts = 10e-6
tau = Ncyc/f0
t = arange(-.1*Ts,0.5*Ts,1/(10*f0))
x_PT = A*ss.rect(t-tau/2,tau)*sin(2*pi*f0*t) + A*ss.rect(t-Ts-tau/2,tau)
plot(t*1e6,x_PT)
title(r'CCS with $f_0=10$MHz, $T_s=10\mu$s, $N_{cyc}=10$, and $A = 100$mV')
ylabel(r'Amplitude')
xlabel(r'Time ($\mu$s)')
#axis('off')
grid();
#savefig('ccs.pdf')
```



Frequency Domain: Power Spectrum in dBm

```
[ ]: f0 = 10.0e6
     A = 0.1/2
     Ncyc = 10
     Ts = 10e-6
     Nstart = 0
     Nstop = 401
     n = arange(Nstart,Nstop)
     Xn_ccs = coherent_cosinusoid(f0,A,Ncyc,Ts,Nstop,Nstart)
     Xn_dBm = line_spectra_dBm(n/Ts/1e6,Xn_ccs,floor_dBm = -90)
     title(r'CCS with $f_0=10$MHz, $T_s=10\mu$s, $N_{cyc}=10$, and $A = 100$mV')
     xlabel(r'Frequency (MHz)');
     ylim([-90,-10])
     #xlim([9.5,10.5]);
     #xlim([7.5,12.5]);
     xlim([0,20]);
```



Note the asymmetry due to the negative frequency spectral lines lapping up over the positive frequency spectral lines. If f_0 was a very high microwave frequency, relative to the pulse period T_s this asymmtery would be diminished greatly. Agree?

1.3 Plotting Line Spectra for the Coherent and Non-Coherent Cases

For both a coherent and non-coherent co-sinusoidal signal the function `line_spectra_CS_dBm()` can be used to used to plot the power spectrum in dBm. This will be a one-sided spectrum that

is consistent with the N9914A spectrum display. The sinusoid frequency is supplied is input as `f0` nad `fn` and `Xn` are ..

Alternatively the function `coherent_cosinusoid()` returns the `Xn` coefficients for the specil case of a coherent co-sinusoid pulse train having exactly `Ncyc` sinusoid cycles per pulse. This makes the sinusoid frequency `f0` an integer multiple of the inverse of the pulse train, i.e., $1/\tau$.

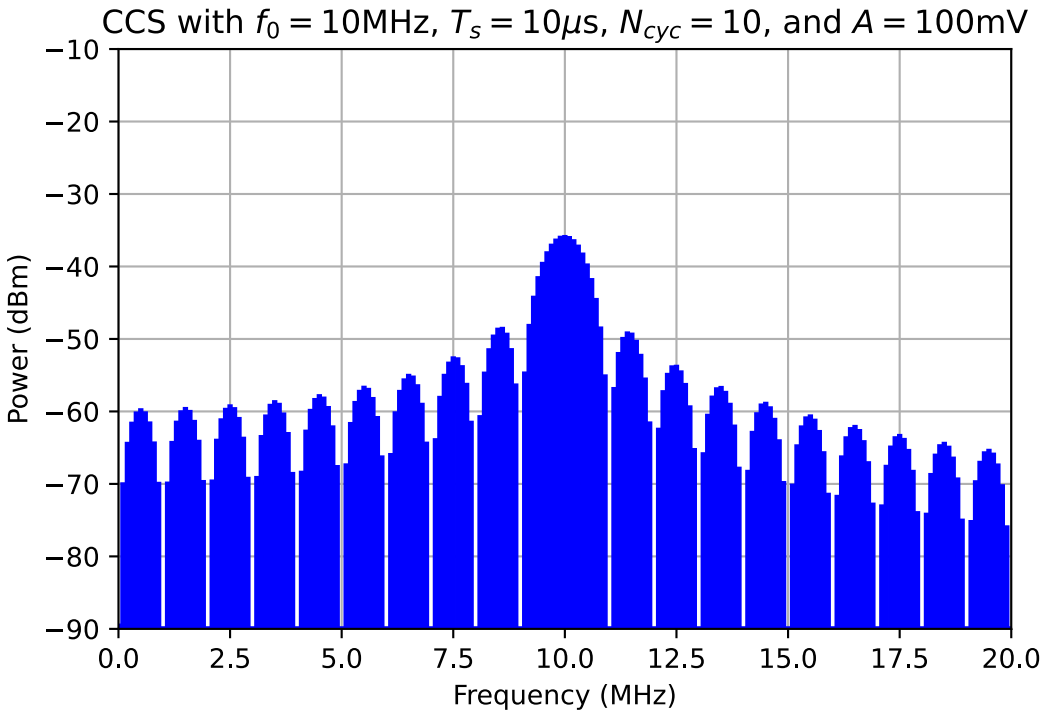
[ ]:
```
def line_spectra_CS_dBm(f0,fn,Xn,floor_dBm = -60,R0 = 50,lm_tol=0.001):
    """
    Plot the one-sided line spectra from Fourier coefficients in dBm for
    bandpass signals, e.g. when a carrier f0 is present. Carrier coherency is
    not required. Overlapping spectral lines folding up from f < 0 will be
    coherently combined if coherence withing line match tolerance lm_tol.

    The dBm values are also returned in an ndarray.

    Inputs
    ------
    f0 = carrier frequency
    fn = harmonic frequencies correspinding to Xn's
    Xn = pulse train FS coefficients
    floor_dBm = spectrum floor in dBm
    R0 = impedance (default 50 ohms)
    lm_tol = line spectra combining match tolerance (default 0.001 in fn units)

    Returns
    -------
    fn_BPu = Unique bandpass line frequencies
    Sx_dBm_BPu = One-sided bandpass lines as dBm levels (combined if possible)
    M_BPu = Multiplicity of line frequencies in fn_BPu (should be 1 or 2)

    Mark Wickert February 2019
    """
    Xn_dBm = abs(Xn*sqrt(1000/(2*R0)))
    #Xn_dBm[0] = abs(Xn[0])*sqrt(1000/(R0))
    fn_BP = hstack((fn[1:]+f0,array([f0]),abs(fn[1:]-f0)))
    Xn_dBm_BP = hstack((Xn_dBm[1:],array([Xn_dBm[0]]),Xn_dBm[1:]))/2
    # Find unique line frequencies as line overlap occurs near DC from
    # negative frequency spectrum. Use phasor addition to combine
    # the overlapping/like spectral lines. lm_tol sets matching tolerance.
    fn_BPu, M_BPu = signal.unique_roots(fn_BP,tol=lm_tol, rtype='avg')
    Xn_dBm_BPu = zeros(len(fn_BPu),dtype = complex128)
    for k in range(len(fn_BPu)):
        idx_fn = np.nonzero(np.ravel(abs(fn_BP - fn_BPu[k])<=lm_tol))[0]
        Xn_dBm_BPu[k] = sum(Xn_dBm_BP[idx_fn])

    ss.line_spectra(fn_BPu,Xn_dBm_BPu,mode='magdB',sides=1,floor_dB=floor_dBm)
    ylabel(r'Power (dBm)')
```

```
    xlabel(r'Frequency (Hz)')
    title(r'Line Spectra PSD')
    Sx_dBm_BPu = 20*log10(2*Xn_dBm_BPu)
    if fn_BPu[0] == 0:
        Sx_dBm_BPu[0] -= 6.02
    return fn_BPu, Sx_dBm_BPu, M_BPu
```

1.3.1 Consider a Pulse train at 1 MHz with a carrier at 10 MHz

Set the pulse width to 100 ns and the peak amplitude of the pulse train into a 50 ohm load of 100 mV.

[ ]:
```
n = arange(0,200)
f0 = 10e6
Ts = 10e-6
tau = 10/f0 # Ncyc/fo
A = 0.1/2 # For the 33600 the sinusoial burst is typically Vp-p => Vpeak = Vp-p/
 ↪2
Xn_cs = A*tau/Ts*sinc(n*tau/Ts)*exp(-1j*2*pi*n*tau/2/Ts) # Pulse train Xn's
fn_u, Sx_dBm_u, M_u = line_spectra_CS_dBm(f0/1e6,n/Ts/1e6,Xn_cs,floor_dBm = -90)
title(r'CCS with $f_0=10$MHz, $T_s=10\mu$s, $N_{cyc}=10$, and $A = 100$mV')
xlabel(r'Frequency (MHz)');
ylim([-90,-10]);
#xlim([7.5,12.5]);
xlim([0,20]);
```



1.4 M Sequence Generator at the Bit Level

Table 1: m-Sequence Generator Feedback Connections for 2 - 15 stages.

Stages	Connections	Stages	Connections
2	$Q_1 \oplus Q_2$	9	$Q_5 \oplus Q_9$
3	$Q_2 \oplus Q_3$	10	$Q_7 \oplus Q_{10}$
4	$Q_3 \oplus Q_4$	11	$Q_9 \oplus Q_{11}$
5	$Q_3 \oplus Q_5$	12	$Q_2 \oplus Q_{10} \oplus Q_{11} \oplus Q_{12}$
6	$Q_5 \oplus Q_6$	13	$Q_1 \oplus Q_{11} \oplus Q_{12} \oplus Q_{13}$
7	$Q_6 \oplus Q_7$	14	$Q_2 \oplus Q_{12} \oplus Q_{13} \oplus Q_{14}$
8	$Q_2 \oplus Q_3 \oplus Q_4 \oplus Q_8$	15	$Q_{14} \oplus Q_{15}$

A Python Class for Bit Level M-Sequences The Python code for a `bit_wise` PN code generator *class*.

[ ]:
```
class bitwise_PN(object):
    """
    Implement a PN generator using bitwise manipulation for
    the shift register. The LSB holds b0 and bits are shifted left.
              +----+----+----+----+----+----+----+
         sr = |bM-1| .. |bM-k| .. | b2 | b1 | b0 |
              +----+----+----+----+----+----+----+
                 |         |
    Feedback:(tap1-1)   (tap2-1)    Shift left using <<

    Mark Wickert February 2017
    """
    def __init__(self,tap1,tap2,Nstage,sr_initialize):
        """
        Initialize the PN generator object
        """
        self.tap1 = tap1 - 1
        self.tap2 = tap2 - 1
        self.mask1 = 0x1 << (tap1 - 1) # to select bit of interest
        self.mask2 = 0x1 << (tap2 - 1) # to select bit of interest
        self.Nstage = Nstage
        self.period = 2**Nstage - 1
        self.sr = sr_initialize
        self.bit = 0
        self.sync_bit = 0

    def clock_PN(self):
        '''
        Method to advance m-sequence generator by one bit
```

```
        XOR tap1 and tap2 SR values and feedback to input
        '''
        fb = ((self.sr & self.mask1)>> self.tap1) ^ \
            ((self.sr & self.mask2) >> self.tap2)
        self.sr = (self.sr << 1) + fb
        self.sr = self.sr & self.period # set MSBs > Nstage to 0
        self.bit = self.sr & 0x1 # output LSB from SR
        # See if all 1's condition exits in SR, if so output a synch pulse
        if ((self.sr & self.period) == self.period):
            self.sync_bit = 0x1
        else:
            self.sync_bit = 0x0
        print('output = %d, sr contents = %s, sync bit = %d' \
             % (self.bit, binary(self.sr, self.Nstage), self.sync_bit))


# A simple binary format display function which shows
# leading zeros to a fixed bit width
def binary(num, length=8):
    return format(num, '#0{}b'.format(length + 2))
```

[ ]:
```
# Create an instance object of the PN class
PN1 = bitwise_PN(10,7,10,0x1)
```

[ ]:
```
PN1.clock_PN()
```

```
output = 0, sr contents = 0b0000000010, sync bit = 0
```

[ ]:
```
# Set the shift register initial condition
sr = 0b1
```

[ ]:
```
# Look at some output values
Nout = 10
x_out = zeros(Nout)
s_out = zeros(Nout)
PN1 = bitwise_PN(3,2,3,0x1)
for k in range(Nout):
    PN1.clock_PN()
    x_out[k] = PN1.bit
    s_out[k] = PN1.sync_bit
```

```
output = 0, sr contents = 0b010, sync bit = 0
output = 1, sr contents = 0b101, sync bit = 0
output = 1, sr contents = 0b011, sync bit = 0
output = 1, sr contents = 0b111, sync bit = 1
output = 0, sr contents = 0b110, sync bit = 0
output = 0, sr contents = 0b100, sync bit = 0
```

```
output = 1, sr contents = 0b001, sync bit = 0
output = 0, sr contents = 0b010, sync bit = 0
output = 1, sr contents = 0b101, sync bit = 0
output = 1, sr contents = 0b011, sync bit = 0
```

1.5 Creating ARB Files for the Keysight 33600A

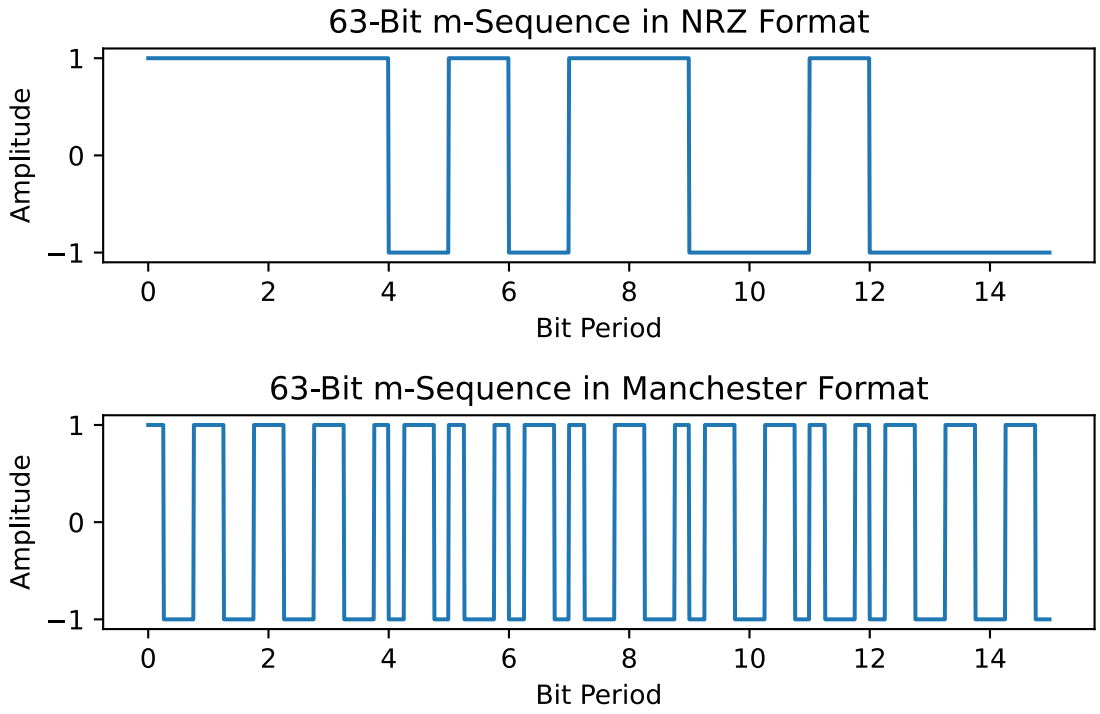
To create an arbitrary (ARB) waveform you fundamentally create a single column file. This can be done in Excel, can be a campture from a scope CSV. Here we will use Python to write the file. The signal values stored in the file correspond to sample values relative to a sampling rate set in the generator. The sampled waveform requirements are:

- A minimum of 32 points
- A maximum of 1,000,000 points
- A maximum sampling rate of 250 MHz
- Waveforms automatically repeat

In this lab we desire a PRBS waveform with NRZ and Manchester *line coding*.

[ ]:
```
# Desire Rb = 1 Mbps with 100x oversampling
fs = 100e6 # this parameter is entered on the 33600A
# Samples per bit
Ns = 100
# One sample per bit M = 63 bit long m-sequence
N_STAGE = 4
# Use the PN_gen (M-seq) function ss over the bitwise_PN
PN_63 = ss.pn_gen(2**N_STAGE - 1, N_STAGE)
# Upsample and make rectangular pulse shaped bits bipolar
x_NRZ = signal.lfilter(ones(Ns),1,ss.upsample(2*PN_63-1,Ns))

# Create an Ns period squarewave clock to convert NRZ to Manchester
n = arange(len(x_NRZ))
x_CLK = sign(cos(2*pi*1e6/fs*n))
# Multiply the clock by the NRZ waveform
x_MAN = x_NRZ*x_CLK

# Optional scaling of amplitudes for fixed-point input
# Internally the 33600A uses 16-bit signed integers, so a reasonable
# full-scale is +/-32000
# x_NRZ *= 32000
# x_MAN *= 32000

subplot(211)
plot(n/Ns,x_NRZ)
title(r'63-Bit m-Sequence in NRZ Format')
ylabel(r'Amplitude')
xlabel(r'Bit Period')
subplot(212)
plot(n/Ns,x_MAN)
```

```
title(r'63-Bit m-Sequence in Manchester Format')
ylabel(r'Amplitude')
xlabel(r'Bit Period')
tight_layout()
```



File Save

- Save the waveforms as a CSV file for input to the Keysight 33600A

[ ]:
```
savetxt('NRZ.csv', x_NRZ, delimiter=',')
savetxt('MAN.csv', x_MAN, delimiter=',')
```

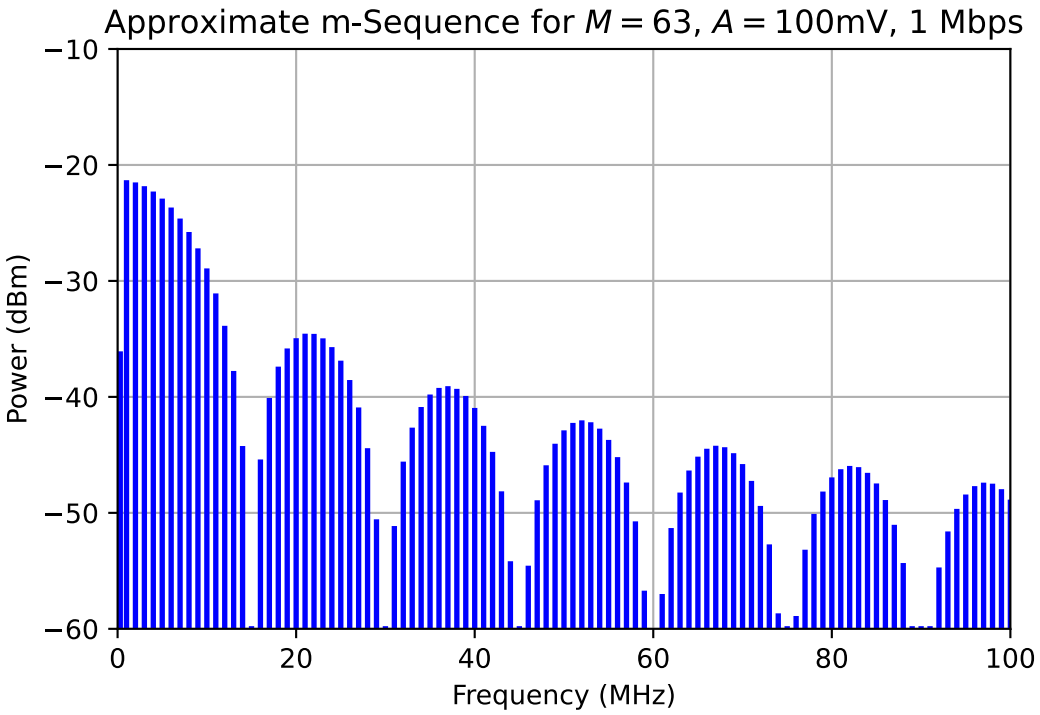
1.5.1 Approximating the Fourier Coefficients for the m-Sequence

Problem 3 has you obtain the exact power spectral density by direct Fourier transform of the the autocorrelation function. An alternative is to obtain the approximate values of the Fourier series coefficients from one period of the waveform. We can use the waveforms generated at 100 samples per bit for the 33600A ARB.

[ ]:
```
Xk, fk = ss.fs_coeff(x_NRZ*.1/2,100,1e6) # NRZ
#Xk, fk = ss.fs_coeff(x_MAN*.1/2,100,1e6) # Manchester
Xn_dBm = line_spectra_dBm(fk/1e6,Xk,-60)
title(r'Approximate m-Sequence for $M=63$, $A = 100$mV, 1 Mbps')
xlabel(r'Frequency (MHz)');
```

```
ylim([-60,-10])
xlim([0,100]);
```

```
/var/folders/58/hy23wcvd6t1g3s0djl92ptbm0000gn/T/ipykernel_96095/2088212032.py:2
5: RuntimeWarning: divide by zero encountered in log10
  Sx_dBm = 20*log10(2*Xn_dBm)
```



1.6 Lab Problem Workspace

Use the space below to create your solutions to the lab problems. As a rule compare theory to measurement when ever possible.

1.6.1 Pulse Train

1.6.2 Part a

Basic pulse train ...

1.6.3 Part b

Pulse train rise and fall time

...

Analysis to Support Finite Rise Time Pulse Train Here we model the finite risetime of a singke pulse as the difference between two triangle pulse shapes. In the frequency domain it will

be the difference between two $\text{sinc}^2()$ functions. I commented to some lab teams that you can get a trapezoid pulse shape by convolving two rectangles, which is true, but to get the desired degrees of freedom, both risetime and pulse width, I found the triangle approach more statisfying.

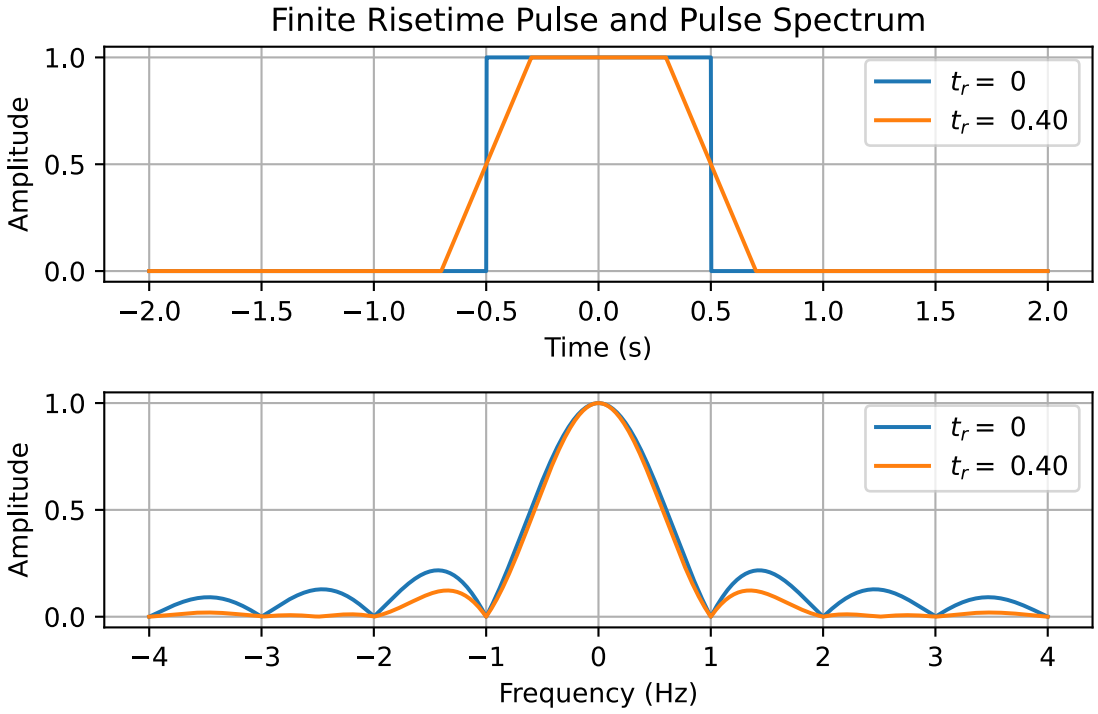
To move things along and give you the opportunity to compare theory with measured, below are two functions. The first creates the time-domain pulse shape (centered on $t = 0$) $p_{\text{finite rise}}(t, \tau, t_r)$ and the second produces the corresponding Fourier transform $P_{\text{finite rise}}(f, \tau, t_r)$.

To compare with your measured spectrum analyzer results you will need to sample the continuous Foujrier transform results given here.

[ ]:
```
# Time domain and frequency domain function for a finite risetime rectangular␣
 ↪pulse
import numpy as np
def p_finite(t,tau,tr):
    """
    Finite Risetime Pulse
    as the difference between two triangle pulses

    p_waveform = p_finite(t,tau,tr)

    Mark Wickert February 2022
    """
    p = np.zeros(len(t))
    T1 = tau - tr
    if tr != 0:
        x1 = (1 + T1/(2*tr))*ss.tri(t,tr+T1/2) # big triangle
        x2 = T1/(2*tr)*ss.tri(t,T1/2) # chop off the top of the big triangle
        p = x1-x2
    else:
        p = ss.rect(t,tau)
    return p

def P_finite(f,tau,tr):
    """
    Finite Risetime Pulse

    P_spectrum = P_finite(f,tau,Tr)

    Mark Wickert February 2022
    """
    P = np.zeros(len(f))
    T1 = tau - tr
    if tr != 0:
        P = (1 + T1/2/tr)*(tr+T1/2)*np.sinc(f*(tr+T1/2))**2 - (T1/2/tr)*T1/2*np.
 ↪sinc(f*T1/2)**2
    else:
        P = T1*np.sinc(f*T1)
```

```
    return P
```

[ ]:
```
t = arange(-2,2,.001)
tau = 1
tr = 0.4
subplot(211)
plot(t,abs(p_finite(t,tau,0)))
plot(t,abs(p_finite(t,tau,tr)))
title(r'Finite Risetime Pulse and Pulse Spectrum')
xlabel(r'Time (s)')
ylabel(r'Amplitude')
legend((r'$t_r =$ 0',r'$t_r =$ %2.2f' % (tr,)))
grid()

f = arange(-4,4,.001)
subplot(212)
plot(f,abs(P_finite(f,tau,0)))
plot(f,abs(P_finite(f,tau,tr)))
xlabel(r'Frequency (Hz)')
ylabel(r'Amplitude')
legend((r'$t_r =$ 0',r'$t_r =$ %2.2f' % (tr,)))
grid()
tight_layout()
```



Interactive Study of Risetime on a Rectangular Pulse Spectrum

[ ]:
```
p_tr = widgets.FloatSlider(value=0.,min=0,max=0.8,step=0.
↪01,description='risetime/tau')
uiV1 = widgets.VBox([p_tr])
ui = widgets.HBox([uiV1])

def f_SPEC(p_tr):

    f = arange(-4,4,.001)
    plt.figure(1)
    # plt.plot(f,abs(P_finite(f,1.0,0)))
    # plt.plot(f,abs(P_finite(f,1.0,p_tr)))
    plt.plot(f,20*log10(abs(P_finite(f,1.0,0))))
    plt.plot(f,20*log10(abs(P_finite(f,1.0,p_tr))))
    plt.ylabel(r'Magnitude (dB)')
    plt.xlabel(r'Normalized Frequency $f\times\tau$')
    plt.title(r'Finite Rise Time Pulse Spectrum')
    plt.legend((r'tr=0',r'tr>0'))
    plt.ylim([-60,2])
    plt.xlim([-4,4])
    plt.grid();
    plt.show()

output = widgets.interactive_output(f_SPEC,{'p_tr': p_tr})
display(ui, output)
```

```
HBox(children=(VBox(children=(FloatSlider(value=0.0, description='risetime/tau',␣
↪max=0.8, step=0.01),)),))
```

```
Output()
```

1.6.4 Cosinusoidal Pulse Train

[ ]:

1.6.5 Pseudo-Random/m-Sequences: PN/PRBS

Here we use the internal pseudo-random binary sequence (PRBS) generator capability of the Keysight 33600A.

[ ]:

1.6.6 Line Coding Using ARM Capability

NRZ and Manchester encoding from NRZ encoding is now possible using the 33600A ARB capability.

[ ]: