scikit-dsp-comm Documentation

Release 2.0.1
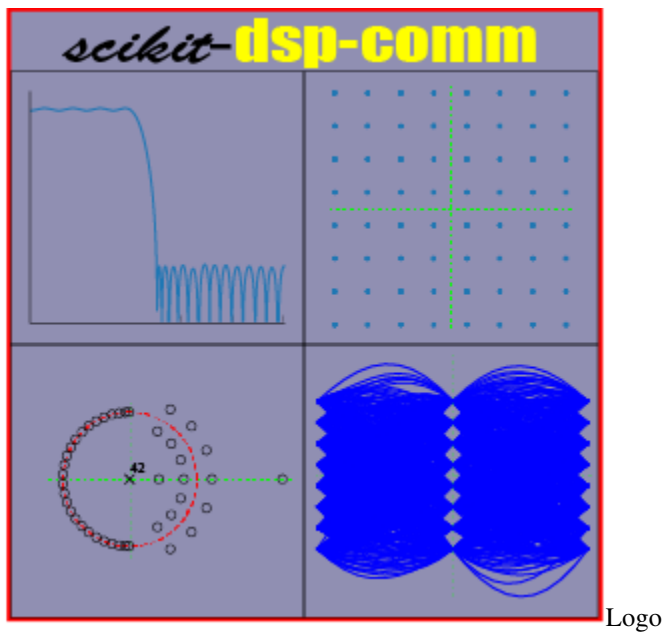
Mark Wickert, Chiranth Siddappa

Nov 09, 2021

CONTENTS

README

Logo

1.1 scikit-dsp-comm

pypi Anaconda-Server Badge Docs

1.1.1 Background

The origin of this package comes from the writing the book Signals and Systems for Dummies, published by Wiley in 2013. The original module for this book is named `ssd.py`. In `scikit-dsp-comm` this module is renamed to `sigsys.py` to better reflect the fact that signal processing and communications theory is founded in signals and systems, a traditional subject in electrical engineering curricula.

1.1.2 Package High Level Overview

This package is a collection of functions and classes to support signal processing and communications theory teaching and research. The foundation for this package is `scipy.signal`. The code in particular currently requires Python `>=3.5x`.

There are presently ten modules that make up scikit-dsp-comm:

1. `sigsys.py` for basic signals and systems functions both continuous-time and discrete-time, including graphical display tools such as pole-zero plots, up-sampling and down-sampling.

2. `digitalcomm.py` for digital modulation theory components, including asynchronous resampling and variable time delay functions, both useful in advanced modem testing.

3. `synchronization.py` which contains phase-locked loop simulation functions and functions for carrier and phase synchronization of digital communications waveforms.

4. `fec_conv.py` for the generation rate one-half and one-third convolutional codes and soft decision Viterbi algorithm decoding, including soft and hard decisions, trellis and trellis-traceback display functions, and puncturing.

5. `fir_design_helper.py` which for easy design of lowpass, highpass, bandpass, and bandstop filters using the Kaiser window and equal-ripple designs, also includes a list plotting function for easily comparing magnitude, phase, and group delay frequency responses.

6. `iir_design_helper.py` which for easy design of lowpass, highpass, bandpass, and bandstop filters using scipy.signal Butterworth, Chebyshev I and II, and elliptical designs, including the use of the cascade of second-order sections (SOS) topology from scipy.signal, also includes a list plotting function for easily comparing of magnitude, phase, and group delay frequency responses.

7. `multirate.py` that encapsulate digital filters into objects for filtering, interpolation by an integer factor, and decimation by an integer factor.

8. `coeff2header.py` write C/C++ header files for FIR and IIR filters implemented in C/C++, using the cascade of second-order section representation for the IIR case. This last module find use in real-time signal processing on embedded systems, but can be used for simulation models in C/C++.

Presently the collection of modules contains about 125 functions and classes. The authors/maintainers are working to get more detailed documentation in place.

1.1.3 Documentation

Documentation is now housed on `readthedocs` which you can get to by clicking the docs badge near the top of this `README`. Example notebooks can be viewed on GitHub pages. In time more notebook postings will be extracted from Dr. Wickert's Info Center.

1.1.4 Getting Set-up on Your System

The best way to use this package is to clone this repository and then install it.

```
git clone https://github.com/mwickert/scikit-dsp-comm.git
```

There are package dependencies for some modules that you may want to avoid. Specifically these are whenever hardware interfacing is involved. Specific hardware and software configuration details are discussed in wiki pages.

For Windows users `pip` install takes care of almost everything. I assume below you have Python on your path, so for example with Anaconda, I suggest letting the installer set these paths up for you.

Editable Install with Dependencies

With the terminal in the root directory of the cloned repo perform an editable `pip` install using

```
pip install -e .
```

Why an Editable Install?

The advantage of the editable `pip` install is that it is very easy to keep `scikit-dsp-comm` up to date. If you know that updates have been pushed to the master branch, you simply go to your local repo folder and

```
git pull origin master
```

This will update you local repo and automatically update the Python install without the need to run `pip` again. **Note**: If you have any Python kernels running, such as a Jupyter Notebook, you will need to restart the kernel to insure any module changes get reloaded.

EXAMPLES

- SciPy 2017 Tutorial

2.1 Jupyter Notebook Examples

```
[1]: %pylab inline
     import sk_dsp_comm.sigsys as ss
     import scipy.signal as signal
     from IPython.display import Image, SVG
```

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

```
[2]: pylab.rcParams['savefig.dpi'] = 100 # default 72
     %config InlineBackend.figure_formats=['svg'] # SVG inline viewing
```

2.1.1 Introduction to Python and the Jupyter Notebook

```
[3]: t = arange(-4,4,.01)
     x = cos(2*pi*t)
     plot(t,x)

     grid()
```

2.1.2 Rectangle and Triangle Pulses Defined

Before showing more examples, consider some familiar signal primitives in your signals and systems background.

To see these defined in the text see in particular Appendix F.5 (p.727) in the table of Fourier transform pairs.

Rectangle

$$\Pi\left(\frac{t}{\tau}\right) = \begin{cases} 1, & |t| \leq \tau/2 \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

Triangle

$$\Lambda\left(\frac{t}{\tau}\right) = \begin{cases} 1 - |t/\tau|, & |t| \leq \tau \\ 0, & \text{otherwise} \end{cases} \tag{2.2}$$

To more readily play with these function represent them numerically in Python. The module `ss.py` has some waveform primitives to help.

```
[4]: t = arange(-5,5,.01)
     x_rect = ss.rect(t-3,2)
     x_tri = ss.tri(t+2,1.5)
     subplot(211)
     plot(t,x_rect)
     grid()
     ylabel(r'$\Pi((t-3)/2)$');
     subplot(212)
     plot(t,x_tri)
     grid()
     xlabel(r'Time (s)')
     ylabel(r'$\Lambda((t+2)/1.5)$');
     tight_layout()
```

- Consider an interactive version of the above:

```
[5]: # Make an interactive version of the above
     from ipywidgets import interact, interactive

     def pulses_plot(D1,D2,W1,W2):
         t = arange(-5,5,.01)
         x_rect = ss.rect(t-D1,W1)
         x_tri = ss.tri(t-D2,W2)
         subplot(211)
         plot(t,x_rect)
         grid()
         ylabel(r'$\Pi((t-3)/2)$');
         subplot(212)
         plot(t,x_tri)
         grid()
         xlabel(r'Time (s)')
         ylabel(r'$\Lambda((t+2)/1.5)$');
         tight_layout()

     interactive_plot = interactive(pulses_plot,D1 = (-3,3,.5), D2 = (-3,3,.5), W1 = (0.5,2,.
     →25), W2 = (0.5,2,.25));
     output = interactive_plot.children[-1]
     output.layout.height = '350px'
     interactive_plot
```

```
interactive(children=(FloatSlider(value=0.0, description='D1', max=3.0, min=-3.0, step=0.
→5), FloatSlider(value...
```

More Signal Plotting

The basic pulse shapes (primitives) defined in the module `ssd.py` are very useful for working Text 2.13a &d, but there are also times when you need a custom piecewise function.

Simple Cases:

Consider plotting

- $x_1(t) = \sin(2\pi \cdot 5t)\Pi((t-2)/2)$ for $0 \leq t \leq 10$
- $x_2(t) = \sum_{n=-\infty}^{\infty} = \Pi((t-5n)/1)$ for $-10 \leq t \leq 10$

```
[6]: t1 = arange(0,10+.01,.01) # arange stops one step size less than the upper limit
     x1 = sin(2*pi*5*t1)* ss.rect(t1-2,2)
     subplot(211)
     plot(t1,x1)
     xlabel(r'Time (s)')
     ylabel(r'$x_1(t)$')
     grid()
     t2 = arange(-10,10,.01)
     # Tweak mod() to take on negative values
     x2 = ss.rect(mod(t2+2.5,5)-2.5,1)
     subplot(212)
     plot(t2,x2)
     xlabel(r'Time (s)')
     ylabel(r'$x_2(t)$')
     grid()
     tight_layout()
```

Custom Piecewise:

A custom piecewise function is a direct and to the point way of getting a more complex function plotted. Consider plotting:

$$x_3(t) = \begin{cases} 1 + t^2, & 0 \leq t \leq 3 \\ \cos(2\pi \cdot 5 \cdot t) & 3 < t \leq 5 \\ 0, & \text{otherwise} \end{cases} \tag{2.3}$$

for $-2 \leq t \leq 6$.

```
[7]: def x3_func(t):
         """
         Create a piecewise function for plotting x3
         """
         x3 = zeros_like(t)
         for k,tk in enumerate(t):
             if tk >= 0 and tk <= 3:
                 x3[k] = 1 + tk**2
             elif tk > 3 and tk <= 5:
                 x3[k] = cos(2*pi*5*tk)
         return x3
```

```
[8]: t3 = arange(-2,6+.01,.01)
     x3 = x3_func(t3)
     plot(t3,x3)
     xlabel(r'Time (s)')
     ylabel(r'$x_3(t)$')
     xlim([-2,6])
     grid()
```

```
[9]: 26/2
```

```
[9]: 13.0
```

2.1.3 Energy and Power Signals

The general definitions are:

$$E \triangleq \lim_{T \to \infty} \int_{-T}^{T} |x(t)|^2 \, dt = \int_{-\infty}^{\infty} |x(t)|^2 \, dt \tag{2.4}$$

$$P \triangleq \lim_{T \to \infty} \frac{1}{2T} \int_{-T}^{T} |x(t)|^2 \, dt \tag{2.5}$$

For the case of a periodic signal, you can take the definition of P above and reduce the calculation down to

$$P = \frac{1}{T} \int_{t_0}^{t_0+T} |x(t)|^2 \, dt \tag{2.6}$$

where t_0 can be any convenient value.

Consider the waveform of Text problem 2.14b

$$x_2(t) = \sum_{n=-\infty}^{\infty} \Lambda\left(\frac{t - 3n}{2}\right) \tag{2.7}$$

You can create an approximation to the waveform over a finite number of periods by doing a little programming:

```
[10]: def periodic_tri(t,tau,T,N):
          """
          Approximate x2(t) by running the sum index from -N to +N.
          The period is set by T and tau is the tri pulse width
          parameter (base width is 2*tau).

          Mark Wickert January 2015
          """
          x = zeros_like(t)
          for n in arange(-N,N+1):
              x += ss.tri(t-T*n,tau)
          return x
```

```
[11]: t = arange(-10,10,.001)
      x = periodic_tri(t,2,6,10)
      plot(t,x)
      plot(t,abs(x)**2)
```

(continues on next page)

```
grid()
#xlim([-5,5])
xlabel(r'Time (s)')
ylabel(r'$x_2(t)$ and $x_2^2(t)$');
```

For the power calculation create a time array that runs over exactly one period. Below is the case for the original problem statement.

```
[12]: T0 = 6
      tp = arange(-T0/2,T0/2+.001,.001)
      xp = periodic_tri(tp,2,T0,5)
      plot(tp,xp)
      plot(tp,abs(xp)**2)
      legend((r'$x(t)$', r'$|x(t)|^2$'),loc='best',shadow=True)
      grid();
      xlim([-T0/2,T0/2])
      xlabel(r'Time (s)')
      ylabel(r'$x_2(t)$ and $x_2^2(t)$');
```

A simple numerical approximation to the integral

$$P = \frac{1}{T} \int_0^T |x_b(t)|^2 \, dt \tag{2.8}$$

is shown below:

```
[13]: #Power calculation
      Px2 = (1/T0)*sum(xp**2)*.001 # rectangular partitions for integral
      print('Power estimate via numerical integration: %2.4f W' % Px2)
```

```
Power estimate via numerical integration: 0.2222 W
```

Power in the Sum of Two Sinusoids

The problem is what is the power in the signal

$$x(t) = A_1 \cos(\omega_1 t + \phi_1) + A_2 \cos(\omega_2 t + \phi_2), \ -\infty < t < \infty \tag{2.9}$$

Since we are not certain that $x(t)$ is periodic, the power calculation requires that we form

$$P_x = \lim_{T \to \infty} \frac{1}{T} \int_{-T/2}^{T/2} |x(t)|^2 \, dt = \langle |x(t)|^2 \rangle \tag{2.10}$$

- Rather that just jumping in and making a mess, consider first the expansion of $|x(t)|^2 = x^2(t)$:

$$x^2(t) = \frac{A_1^2}{2} \left[1 + \cos(2\omega_1 t + \phi_1) \right] + \frac{A_2^2}{2} \left[1 + \cos(2\omega_2 t + \phi_2) \right] \tag{2.11}$$

$$+ 2\frac{A_1 A_2}{2} \left\{ \cos[(\omega_1 + \omega_2)t + (\phi_1 + \phi_2)] + \cos[(\omega_1 - \omega_2)t + (\phi_1 - \phi_2)] \right\} \tag{2.12}$$

- The time average operator is linear, so we consider $\langle \ \rangle$ operating on each term of the above independently

- For $\omega_1 \neq \omega_2$, the first two terms yield $A_1^2/2$ and $A_2^2/2$ respectively

---

- The last term requires some thinking, but as long as $\omega_1 \neq \omega_2$ the times average of $\cos[(\omega_1 + \omega_2)t + (\phi_1 + \phi_2)]$ and $\cos[(\omega_1 - \omega_2)t + (\phi_1 - \phi_2)]$, the two terms respectively are each zero!

- Finally,

$$P_x = \frac{A_1^2}{2} + \frac{A_2^2}{2} \tag{2.13}$$

- When the frequencies are equal, then you can combine the terms using trig identities (recall the phasor addition formula from ECE 2610

$$x(t) = A\cos(\omega t + \phi) \tag{2.14}$$

where $\omega = \omega_1 = \omega_2$ and

$$Ae^{j\phi} = A_1 e^{j\phi_1} + A_2 e^{j\phi_2} \tag{2.15}$$

```
[14]: t = arange(-10,10,.001)
      x1 = 4*cos(2*pi*10*t)
      x2 = 3*cos(2*pi*3.45*t+pi/9)
      plot(t,x1)
      plot(t,x2)
      plot(t,x1+x2)
      grid()
      xlabel(r'Time (s)')
      ylabel(r'Amplitude')
      legend((r'$x_1(t)$', r'$x_2(t)$', r'$x_1(t)+x_2(t)$'),loc='best',shadow=True)
      xlim([-.1,.1]);
```

```
[15]: print('Power calculations: %3.2f, %3.2f, %3.2f' \
            % (var(x1),var(x2),var(x1+x2)))
```

```
Power calculations: 8.00, 4.50, 12.50
```

```
[16]: print('Theory: %3.2f, %3.2f, %3.2f' \
            % (4**2/2,3**2/2,4**2/2+3**2/2))
```

```
Theory: 8.00, 4.50, 12.50
```

2.1.4 Fourier Series and Line Spectra Plotting

Being able to easily plot the line spectra of periodic signals will hopefully enhance your understanding. The module `ss.py` contains the function `ss.line_spectra()` for this purpose. The function assumes that the Fourier coefficients, X_n are available for a real signal $x(t)$. The function plots line spectra as: * The two-sided magnitude spectra * The two-sided magnitude spectra in dB with an adjustable floor level in dB * The two-sided phase spectra in radians * The one-sided line spectra corresponding to the three cases listed immediately above Examples are given below for the case of a simple pulse train and then for a trapezoidal pulse train. IN the case of the trapezoidal pulse train the underlying Fourier coefficients are obtained numerically using the FFT as described in the course notes.

A fundamental requirement in using `ss.line_spectra()` is to beable to supply the coefficients starting with the DC term coefficient X_0 and moving up to the Nth harmonic. Before plotting the pulse train line spectra I first describe a *helper* function for visualizing the pulse train waveform.

Pulse Train

```
[17]: def pulse_train(Np,fs,tau,t0):
          """
          Generate a discrete-time approximation to a continuous-time
          pulse train signal. Amplitude values are [0,1]. Scale and offset
          later if needed.

          Inputs
          ------
           Np = number of periods to generate
           fs = samples per period
          tau = duty cycle
           t0 = pulse delay time relative to first rising edge at t = 0

          Return
          ------
          t = time axis array
          x = waveform

          Mark Wickert, January 2015
          """
          t = arange(0,Np*fs+1,1)/fs #time is normalized to make period T0 = 1.0
          x = zeros_like(t)
          # Using a brute force approach, just fill x with the sample values
          for k,tk in enumerate(t):
              if mod(tk-t0,1) <= tau and mod(tk-t0,1) >= 0:
                  x[k] = 1
          return t,x
```

```
[18]: tau = 1/8; fs = 8*16; t0 = 0 # note t0 = tau/2
      subplot(211)
      t,x = pulse_train(4,fs,tau,t0)
      plot(t,x) # Just a plot of xa(t)
      ylim([-.1,1.1])
      grid()
      ylabel(r'$x_a(t)$')
      title(r'Pulse Train Signal: (top) $x_a(t)$, (bot) $x_b(t) = 1-x_a(t)$');
      subplot(212)
      t,x = pulse_train(4,fs,tau,t0)
      plot(t,1-x) # Note here y(t) = 1 - x(t), a special case of
      ylim([-.1,1.1]) # y(t) = A + B*x(t) in the notes
      grid()
      xlabel(r'Time ($t/T_0$)')
      ylabel(r'$x_b(t)$');
```

Example: Pulse Train Line Spectra

For the case of pulse train having the initial pulse starting at $t = 0$, i.e.,

$$x(t) = \sum_{k=-\infty}^{\infty} A \cdot \Pi \left(\frac{t - \tau/2 - kT_0}{\tau} \right), \tag{2.16}$$

the Fourier coefficient are given by

$$X_n = A \cdot \frac{\tau}{T_0} \cdot \text{sinc}(nf_0\tau) \cdot \exp(-j2\pi nf_0t_0) \tag{2.17}$$

where $f_0 = 1/T_0$ is the fundamental frequency and here $t_0 = \tau/2$.

Line spectra plotting is shown below for this case. If the pulse train should be shifted in time to some other orientation, then the phase plot will change, as the included $\exp(j2\pi nf_0t_0)$ term will be different.

Note: The pulse train function define above is slightly different from the pulse train defined in the book and shown in mathematical form as $x(t)$ just above in this cell. The function `pulse_train()` has the first pulse starting exactly at $t = 0$. To move the pule train right or left on the time axis, you can use the function parameter `t0`.

```
[19]: n = arange(0,25+1) # Get 0 through 25 harmonics
      tau = 0.125; f0 = 1; A = 1;
      Xn = A*tau*f0*sinc(n*f0*tau)*exp(-1j*2*pi*n*f0*tau/2)
      # Xn = -Xn # Convert the coefficients from xa(t) t0 xb(t)
      # Xn[0] += 1
      figure(figsize=(6,2))
      f = n # Assume a fundamental frequency of 1 Hz so f = n
      ss.line_spectra(f,Xn,mode='mag',sides=2,fsize=(6,2))
      xlim([-25,25]);
      #ylim([-50,10])
      figure(figsize=(6,2))
      ss.line_spectra(f,Xn,mode='phase',fsize=(6,2))
      xlim([-25,25]);
```

```
<Figure size 432x144 with 0 Axes>
```

```
<Figure size 432x144 with 0 Axes>
```

Example: Trapezoidal Pulse

Consider the line spectra of a finite rise and fall time pulse train is of practical interest. The function `trap_pulse()` allows you first visualize one period of the trapezoidal pulse train, and then use this waveform in obtaining numerically the Fourier coefficients of this signal. PLotting the corresponding line spectra follows.

A point to be main is that by slowing down the edges (rise time/fall time) of the pulse train the amplitude of the harmonics falls off more rapidly. When considering the clock speed in todays PCs this can be a good thing as harmonic emission is an issue.

```
[20]: def trap_pulse(N,tau,tr):
          """

          xp = trap_pulse(N,tau,tr)


          Mark Wickert, January 2015
          """
```

```
    n = arange(0,N)
    t = n/N
    xp = zeros(len(t))
    # Assume tr and tf are equal
    T1 = tau + tr
    # Create one period of the trapezoidal pulse waveform
    for k in n:
        if t[k] <= tr:
            xp[k] = t[k]/tr
        elif (t[k] > tr and t[k] <= tau):
            xp[k] = 1
        elif (t[k] > tau and t[k] < T1):
            xp[k] = -t[k]/tr + 1 + tau/tr;
        else:
            xp[k] = 0
    return xp, t
```

Let $\tau = 1/8$ and $t_r = 1/20$:

```
[21]: # tau = 1/8, tr = 1/20
N = 1024
xp,t = trap_pulse(N,1/8,1/20)
Xp = fft.fft(xp)
figure(figsize=(6,2))
plot(t,xp)
grid()
title(r'Spectra of Finite Risetime Pulse Train: $\tau = 1/8$ $t_r = 1/20$')
ylabel(r'$x(t)$')
xlabel('Time (s)')
f = arange(0,N/2)
ss.line_spectra(f[0:25],Xp[0:25]/N,'magdB',floor_dB=-80,fsize=(6,2))
ylabel(r'$|X_n| = |X(f_n)|$ (dB)');
#% tau = 1/8, tr = 1/10
xp,t = trap_pulse(N,1/8,1/10)
Xp = fft.fft(xp)
figure(figsize=(6,2))
plot(t,xp)
grid()
title(r'Spectra of Finite Risetime Pulse Train: $\tau = 1/8$ $t_r = 1/10$')
ylabel(r'$x(t)$')
xlabel('Time (s)')
ss.line_spectra(f[0:25],Xp[0:25]/N,'magdB',floor_dB=-80,fsize=(6,2))
ylabel(r'$|X_n| = |X(f_n)|$ (dB)');
```

With the edge speed slowed down it is clear that the harmonics drop off faster.

2.1.5 Fourier Transforms

The Fourier transfrom definition is:

$$X(f) = \int_{-\infty}^{\infty} x(t) \, e^{-j2\pi ft} \, dt \tag{2.18}$$

$$x(t) = \int_{-\infty}^{\infty} X(f) \, e^{j2\pi ft} \, df \tag{2.19}$$

A numerical approximation to the Fourier transform is possible using the FFT, or more conveniently using the function `freqz()` from the package `scipy.signal`. A helper function to abstract some of the digital signal processing details is `f, X = FT_approx(x,dt,Nfft)`. The function is now part of `sigsys.py` with name change to `ft_approx()`:

```
[22]: def FT_approx(x,t,Nfft):
          '''
          Approximate the Fourier transform of a finite duration
          signal using scipy.signal.freqz()

          Inputs
          ------
             x = input signal array
             t = time array used to create x(t)
          Nfft = the number of frdquency domain points used to
                 approximate X(f) on the interval [fs/2,fs/2], where
                 fs = 1/Dt. Dt being the time spacing in array t

          Return
          ------
          f = frequency axis array in Hz
          X = the Fourier transform approximation (complex)

          Mark Wickert, January 2015
          '''
          fs = 1/(t[1] - t[0])
          t0 = (t[-1]+t[0])/2 # time delay at center
          N0 = len(t)/2 # FFT center in samples
          f = arange(-1/2,1/2,1/Nfft)
          w, X = signal.freqz(x,1,2*pi*f)
          X /= fs # account for dt = 1/fs in integral
          X *= exp(-1j*2*pi*f*fs*t0)# time interval correction
          X *= exp(1j*2*pi*f*N0)# FFT time interval is [0,Nfft-1]
          F = f*fs
          return F, X
```

Example: Rectangular Pulse

As as simple starting point example, consider $x(t) = \Pi(t\tau)$. The well known result for the Fourier transfrom (FT) is:

$$X(f) = \mathcal{F}\left\{\Pi\left(\frac{t}{\tau}\right)\right\} = \tau \, \mathrm{sinc}(f\tau) \tag{2.20}$$

We now use the above defined `FT_approx()` to obtain a numerical approximation to the FT of the rectangular pulse.

Tips: * Make sure the signal is well contained on the time interval used to generate $x(t)$ * Make sure the sampling rate, one over the sample spacing, is adequate to represent the signal spectrum * From sampling theory, the reange of frequencies represented by the spectrum estimate will be $f_s/2 \le f < f_s/2$

```
[23]: fs = 100 # sampling rate in Hz
      tau = 1
      t = arange(-5,5,1/fs)
      x0 = ss.rect(t-.5,tau)
      figure(figsize=(6,5))
      subplot(311)
      plot(t,x0)
      grid()
      ylim([-0.1,1.1])
      xlim([-2,2])
      title(r'Exact Waveform')
      xlabel(r'Time (s)')
      ylabel(r'$x_0(t)$');

      # FT Exact Plot
      fe = arange(-10,10,.01)
      X0e = tau*sinc(fe*tau)
      subplot(312)
      plot(fe,abs(X0e))
      #plot(f,angle(X0))
      grid()
      xlim([-10,10])
      title(r'Exact Spectrum Magnitude')
      xlabel(r'Frequency (Hz)')
      ylabel(r'$|X_0e(f)|$');

      # FT Approximation Plot
      f,X0 = ss.ft_approx(x0,t,4096)
      subplot(313)
      plot(f,abs(X0))
      #plot(f,angle(X0))
      grid()
      xlim([-10,10])
      title(r'Approximation Spectrum Magnitude')
      xlabel(r'Frequency (Hz)')
      ylabel(r'$|X_0(f)|$');
      tight_layout()
```

Example: Text Problem 2.31a Drill Down

In this problem you are given

$$x_1(t) = \Pi\left(\frac{t + 1/2}{1}\right) - \Pi\left(\frac{t - 1/2}{1}\right) \tag{2.21}$$

The Fourier transfrom of this signal can be found using *linearity* and the *time delay* theorems.

$$X_1(f) = \mathcal{F}\left\{\Pi\left(\frac{t + 1/2}{1}\right) - \Pi\left(\frac{t - 1/2}{1}\right)\right\} \tag{2.22}$$

$$= \operatorname{sinc}(f) \cdot \left[e^{j2\pi f \cdot 1/2} - e^{-j2\pi f \cdot 1/2}\right] \times \frac{2j}{2j} \tag{2.23}$$

$$= 2j \operatorname{sinc}(f) \cdot \sin(\pi f) \tag{2.24}$$

2.1. Jupyter Notebook Examples 15

```
[24]: fs = 100
      t = arange(-5,5,1/fs)
      x1 = ss.rect(t+1/2,1)-ss.rect(t-1/2,1)
      subplot(211)
      plot(t,x1)
      grid()
      ylim([-1.1,1.1])
      xlim([-2,2])
      xlabel(r'Time (s)')
      ylabel(r'$x_1(t)$');
      fe = arange(-10,10,.01)
      X1e = 2*1j*sinc(fe)*sin(pi*fe)
      f,X1 = ss.ft_approx(x1,t,4096)
      subplot(212)
      plot(f,abs(X1))
      plot(fe,abs(X1e))
      #plot(f,angle(X1))
      legend((r'Num Approx',r'Exact'),loc='best')
      grid()
      xlim([-10,10])
      xlabel(r'Frequency (Hz)')
      ylabel(r'$|X_1(f)|$');
      tight_layout()
```

- Notice the numerical approximation and exact spectral plots overlay one another

Example: Modulation Theorem

Consider the modulation theorem, which is extremely important to communications theory:

$$y(t) = x(t) \cdot \cos(2\pi f_0 t) \tag{2.25}$$

$$Y(f) = \frac{1}{2} \left[X(f - f_0) + X(f + f_0) \right] \tag{2.26}$$

Here we will use a triangle pulse for $x(t)$:

```
[25]: fs = 100 # sampling rate in Hz
      tau = 1
      t = arange(-5,5,1/fs)
      x3 = ss.tri(t,tau)
      y = x3*cos(2*pi*10*t)
      subplot(211)
      plot(t,x3)
      plot(t,y)
      grid()
      ylim([-1.1,1.1])
      xlim([-2,2])
      legend((r'$x_3(t)$', r'$y(t)$'),loc='lower right',shadow=True)
      title(r'Time Domain: $x_3(t)$ and $y(t)=x_3(t)\cos(2\pi\cdot 5\cdot t)$')
      xlabel(r'Time (s)')
      ylabel(r'$y(t)$');
```

(continues on next page)

```
f,Y = ss.ft_approx(y,t,4096)
subplot(212)
plot(f,abs(Y))
#plot(f,angle(X0))
grid()
title(r'Frequency Domain: $Y(f)$')
xlim([-15,15])
xlabel(r'Frequency (Hz)')
ylabel(r'$|Y(f)|$');
tight_layout()
```

Example: Representing a Bandlimited Signal

We know that in theory a bandlimited signal can only be generated from a signal having infinite duration. Specifically, a signal with rectangular spectrum has Fourier transfrom pair:

$$x(t) = 2W \operatorname{sinc}(2Wt) \overset{\mathcal{F}}{\Leftrightarrow} \Pi\left(\frac{f}{2W}\right) = X(f) \tag{2.27}$$

In a simulation we expect to have troubles modeling the finite duration aspects of the signal.

```
[26]: fs = 100 # sampling rate in Hz
      W = 5
      t = arange(-5,5,1/fs)
      x4 = 2*W*sinc(2*W*t)
      figure(figsize=(6,2))
      plot(t,x4)
      grid()
      #ylim([-1.1,1.1])
      xlim([-2,2])
      title(r'Time Domain: $x_4(t),\ W = 5$ Hz')
      xlabel(r'Time (s)')
      ylabel(r'$x_4(t)$');
      f,X4 = ss.ft_approx(x4,t,4096)
      figure(figsize=(6,2))
      plot(f,abs(X4))
      grid()
      title(r'Frequency Domain: $X_4(f)$')
      xlim([-10,10])
      xlabel(r'Frequency (Hz)')
      ylabel(r'$|X_4(f)|$');
      figure(figsize=(6,2))
      plot(f,20*log10(abs(X4)))
      grid()
      title(r'Frequency Domain: $X_4(f)$ in dB')
      ylim([-50,5])
      xlim([-10,10])
      xlabel(r'Frequency (Hz)')
      ylabel(r'$|X_4(f)|$ (dB)');
```

Note: The dB version (last plot) reveals that the first sidelobes of the spectrum are only down ~21dB. Increasing the length of the time window will not help. The spectral side lobes will become more tightly packed, but the first sidelobe will still be down only 21dB. With other pulse shapes in the time domain, i.e., note simply a truncted sinc() function reduced sidelobes can be obtained.

2.1.6 Convolution

- The convolution of two signals $x_1(t)$ and $x_2(t)$ is defined as

$$x(t) = x_1(t) * x_2(t) = \int_{-\infty}^{\infty} x_1(\lambda)x_2(t-\lambda)\,d\lambda \tag{2.28}$$

$$= x_2(t) * x_1(t) = \int_{-\infty}^{\infty} x_2(\lambda)x_1(t-\lambda)\,d\lambda \tag{2.29}$$

- A special convolution case is $\delta(t - t_0)$

$$\delta(t - t_0) * x(t) = \int_{-\infty}^{\infty} \delta(\lambda - t_0)x(t-\lambda)\,d\lambda \tag{2.30}$$

$$= x(t-\lambda)\big|_{\lambda=t_0} = x(t - t_0) \tag{2.31}$$

You can experiment with the convolution integral numerically using `ssd.conv_integral()` found in the module `ssd.py`.

```
[27]: t = arange(-2,2.001,.001)
      p1 = ss.rect(t,1)
      p2 = ss.rect(t,3)
      y,ty = ss.conv_integral(p1,t,p2,t)
      plot(ty,y)
      ylim([-.01,1.01])
      grid()
      xlabel(r'Time (s)')
      ylabel(r'$y(t)$');
```

For convolutions involving semi-infinite signals, such as $u(t)$, you can tell `ssd.conv_integral()` about this via the optional extent argument. See the function help using

```
ss.conv_integral?
```

```
[28]: # Consider a pulse convolved with an exponential ('r' type extent)
      tx = arange(-1,8,.01)
      x = ss.rect(tx-2,4) # pulse starts at t = 0
      h = 4*exp(-4*tx)*ss.step(tx)
      y,ty = ss.conv_integral(x,tx,h,tx,extent=('f','r')) # note extents set
      plot(ty,y) # expect a pulse charge and discharge waveform
      grid()
      title(r'$\Pi((t-2)/4)\ast 4 e^{-4t} u(t)$')
      xlabel(r'Time (s)')
      ylabel(r'$y(t)$');
```

2.1.7 Spectrum of PN Sequence (exact)

The cell below is a copy of the earlier pulse train line spectra example. Use this as a template to create the solution to the PN code problem of HW 3.

```
[29]: n = arange(0,25+1) # Get 0 through 25 harmonics
      tau = 0.125; f0 = 1; A = 1;
      Xn = A*tau*f0*sinc(n*f0*tau)*exp(-1j*2*pi*n*f0*tau/2)
      # Xn = -Xn # Convert the coefficients from xa(t) t0 xb(t)
      # Xn[0] += 1
      figure(figsize=(6,2))
      f = n # Assume a fundamental frequency of 1 Hz so f = n
      ss.line_spectra(f,Xn,mode='mag',sides=2,fsize=(6,2))
      xlim([-25,25]);
      #ylim([-50,10])
      figure(figsize=(6,2))
      ss.line_spectra(f,Xn,mode='phase',fsize=(6,2))
      xlim([-25,25]);
```

```
<Figure size 432x144 with 0 Axes>
```

```
<Figure size 432x144 with 0 Axes>
```

2.1.8 Spectrum of PN Sequence (approx)

The code below approximates the PSD of the PN code using a numerical approximation to the Fourier coefficients, X_n. This development may be useful for the lab, as you can esily change the waveform level without having to rework the theory.

The approach taken here to create one period of the PN waveform at 10 samples per bit. The line containing the function `ss.upsample()` converts the bit sequence into a waveform by upsampling and filtering with a rectangular pulse shape (`ones(10)`). The function `ss.fs_coeff()` numerically calculates the X_n's. To plot the PSD from the Fourier coefficients we use

$$S_x(f) = \sum_{n=-\infty}^{\infty} |X_n|^2 \delta(f - nf_0)$$

where f_0 in this case is $1/(MT_0)$ with T_0 beging the bit period and M the code period in bits.

```
[30]: x_PN4 = ss.m_seq(4)
      x = signal.lfilter(ones(10),1,ss.upsample(x_PN4,10))
      t = arange(0,len(x))/10
      figure(figsize=(6,2))
      plot(t,x);
      title(r'Time Domain and PSD of $M=15$ PN Code with $T = 1$')
      xlabel(r'Time (s)')
      ylabel(r'x(t)')
      axis([0,15,-0.1,1.1]);
      grid()
      # 10 samples/bit so 150 samples/period
```

```
# harmonics spaced by 1/(15*T) = 1/15
Xk,fk = ss.fs_coeff(x,45,1/15)
ss.line_spectra(fk,Xk,'magdB',lwidth=2.0,floor_dB=-50,fsize=(6,2))
xlim([-3,3])
ylabel(r'$|X_n| = |X(f_n)|$ (dB)');
```

```
[31]: # Line spacing
      1/15
```

```
[31]: 0.06666666666666667
```

```
[32]: import sk_dsp_comm.digitalcom as dc
      y_PN5_bits = ss.pn_gen(10000,5)
      # Convert to waveform level shifted to +/-1 amplitude
      y = 2*signal.lfilter(ones(10),1,ss.upsample(y_PN5_bits,10))-1
      # Find the time averaged autocorrelation function normalized
      # to have a peak amplitude of 1
      Ry,tau = dc.xcorr(y,y,400)
      # We know Ry is real so strip small imag parts from FFT-based calc
      Ry = Ry.real
```

```
[33]: fs = 10
      t = arange(len(y))/fs
      plot(t[:500],y[:500])
      title(r'PN Waveform for 5 Stages (Period $2^5 -1 = 31$ bits)')
      ylabel(r'Amplitude')
      xlabel(r'Bits (10 samples/bit)')
      grid();
```

```
[34]: tau_s = tau/10
      figure(figsize=(6,2))
      plot(tau_s,Ry)
      title(r'Autocorrelation and PSD Estimates for $M=31$ with $T = 1$')
      xlabel(r'Autocorrelation Lag $\tau$ (s)')
      ylabel(r'$R_y(\tau)$')
      grid();
      figure(figsize=(6,2))
      psd(y,2**12,10)
      xlabel(r'Frequency (Hz)')
      ylabel(r'$S_y(f)$ (dB)')
      #xlim([0,.002]);
      ylim([-30,20]);
```

In Lab 2 of ECE 4670 a C/C++ version of a PN generator is implemented to run the ARM mbed LPC 1768 microcontroller (https://www.sparkfun.com/products/9564). At the heart of this code is:

```
// Globals defined as unsigned int
tap1 -= 1;
tap2 -= 1;
```

```
mask1 = 0x1 << (tap1);
mask2 = 0x1 << (tap2);
bit = 0x0;
sync = 0x0;

void gen_PN() {
    my_pin5 = bit;
    my_pin6 = synch_bit;
    led2 = bit;
    led3 = synch_bit;
    if (clk_state == 0x1)
    {
    // Advance m-sequence generator by one bit
    // XOR tap1 and tap2 SR values and feedback to input
        fb = ((sr & mask1)>> tap1) ^ ((sr & mask2) >> tap2);
        sr = (sr << 1) + fb;
        bit = sr & 0x1;
        // Use random number generator in place of m-sequence bits
        if (DIP_sw4)
        {
            bit = rand_int() & 0x1;
        }
        clk_state = 0x0;
        // See if all 1's condition exists in SR
        if ((sr & synch) == synch) {
            synch_bit = 0x1;
        }
        else
        {
            synch_bit = 0x0;
        }
    }
    else
    {
        if (DIP_sw1) bit = !bit;
        clk_state = 0x1;
    }
}
```

The data type is `unsigned int`, which on the mbed is `uint16_t`, an unsigned 16-bit integer. A single unsigned integer is used as a 16-bit shift register with the LSB, furthest bit to the right, used to represent the first register stage. The shift register is advanced using a left shift << bitwise operation. We can code this Python almost directly, as shown below.

```
[35]: 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))
```

```
[36]: # 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))
```

```
[37]: PN1 = bitwise_PN(10,7,10,0x1)
```

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

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

```
[39]: # sr initial condition
      sr = 0b1
```

```
[40]: Nout = 20
```

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

```
[41]: stem(x_out)
      stem(0.2*s_out,markerfmt = 'ro')
      ylim([0,1.1])
```

```
[41]: (0.0, 1.1)
```

Cross Correlation and Signal Delay

The idea of the autocorrelation function can be extended to the cross correlation, that is the correlation or likeness between two signals, say $x(t)$ and $y(t)$. Define

$$R_{xy}(\tau) = \langle x(t)y(t+\tau) \rangle = \lim_{T \to \infty} \frac{1}{2T} \int_{-T}^{T} x(t)y(t+\tau) \, dt \qquad (2.32)$$

Consider a simulation example using `dc.xcorr(x,t,lags)`:

```
[42]: import sk_dsp_comm.digitalcom as dc
      x_PN4_bits = ss.pn_gen(10000,6)
      # Convert to waveform level shifted to +/-1 amplitude
      x_s = 2*signal.lfilter(ones(10),1,ss.upsample(x_PN4_bits,10))-1
      # Form a delayed version of x_S
      T_D = 35 # 35 sample delay
      y_s = signal.lfilter(concatenate((zeros(T_D),array([1]))),1,x_s)
```

```python
figure(figsize=(6,2))
plot(x_s[:200])
plot(y_s[:200])
ylim([-1.1,1.1])
title(r'Delayed and Undelayed Signals for $T_D = 35$ Samples')
xlabel(r'Samples (10/PN bit)')
ylabel(r'$x_s(t)$ and $y_s(t)$')
grid();
# Find the time averaged autocorrelation function normalized
# to have a peak amplitude of 1
Ryx,tau = dc.xcorr(y_s,x_s,200) #note order change
# We know Ryx is real
Ryx = Ryx.real
tau_s = tau/10
figure(figsize=(6,2))
plot(tau_s,Ryx)
title(r'Cross Correlation for $M=4$ with $T = 1$ and Delay 35 Samples')
xlabel(r'Autocorrelation Lag $\tau$ (s)')
ylabel(r'$R_{yx}(\tau)$')
grid();
```

2.1.9 Spectral Containment Bandwidth (text problem 2.55)

In text problem 2.55 you are asked to find the 90% energy contain bandwidth of a signal $x_i(t)$. Specifically you are to find the lowpass or one-sided bandwidth of a baseband signal such that 90% of the total signal energy is contained in the bandwidth, B_{90}. You obtain B_{90} by solving the following equation

$$0.9 = \frac{0.9 E_{\text{total}}}{E_{\text{total}}} = \frac{\int_{-B_{90}}^{B_{90}} G(f)df}{\int_{-\infty}^{\infty} G(f)df} = \frac{2\int_0^{B_{90}} G(f)df}{2\int_0^{\infty} G(f)df} = \frac{\int_0^{B_{90}} G(f)df}{\int_0^{\infty} G(f)df}, \tag{2.33}$$

where $G(f) = |X_i(f)|^2$ is the energy spectral density of $x_i(t)$.

For parts (c) and (d) the problem states you need to perform numerical integration.

Example:

In an exalier example found in this notebook I found the Fourier transform of

$$x(t) = \Pi\left(\frac{t - \tau/2}{\tau}\right) - \Pi\left(\frac{t + \tau/2}{\tau}\right) \tag{2.34}$$

to be

$$X(f) = 2j \operatorname{sinc}(f\tau) \cdot \sin(\pi f\tau) \tag{2.35}$$

Note I have modified the problem to now have pulse width τ to better match the homework problem where τ is a variable.

The energy spectral density is

$$G(f) = 4 \operatorname{sinc}^2(f\tau) \cdot \sin^2(\pi f\tau) \tag{2.36}$$

I convenient way to numerically integrate $G(f)$ is using simple reactangular partitions, but making sure that Δf is small relative to the changes in $G(f)$. Since you do not know what the value of τ you consider a *normalized frequency* variable $f_n = f\tau$ in the analysis. The rest of the steps are:

1. Sweep $G(f_n)$ using an array `fn` running from zero to f_n large enough to insure that $G(f_n)$ is very small relative to it largest value. In Python this is just filling an array, `Gn` with the functional values.

2. Form a new array which contains the cumulative sum of the values in `Gn`, say `Gn_cumsum = cumsum(Gn)`. Aso form the sum of the array values, i.e., `Gn_tot = sum(Gn)`

3. Plot the ratio of `` `Gn_cumsum/Gn_sum `` versus `fn`. The curve should start at zero and climb to one as f_n becomes large. The value of f_n where the curve crosses through 0.9 is the 90% containment bandwidth.

Note: You might notice that Δf, which is needed in the rectangular integration formula was never used. Why? In the calculation of the cumulative sum and the calculation of the total, both should include Δf, hence in the ration the values cancel out. Nice!

```
[43]: fn = arange(0,10,.001)
      Gn = 4*sinc(fn)**2 * sin(pi*fn)**2
      Gn_cumsum = cumsum(Gn)
      Gn_tot = sum(Gn)
      plot(fn,Gn_cumsum/Gn_tot)
      grid()
      xlabel('Normalized Frequency $f\tau$')
      ylabel('Fractional Power Containment');
```

```
[44]: fn_idx = np.nonzero(np.ravel(abs(Gn_cumsum/Gn_tot - 0.9)< 0.0005))[0]
      fn_idx
```

```
[44]: array([1446, 1447, 1448, 1449, 1450])
```

```
[45]: print('The normalized 90 percent containment bandwidth is %2.2f Hz-s.' \
            % fn[1448])
```

```
The normalized 90 percent containment bandwidth is 1.45 Hz-s.
```

2.1.10 Filter Analysis

To facilitate the performance analysis of both discrete-time and continuous-time filters, the functions `freqz_resp()` and `freqs_resp()` are available (definitions below, respectively). With these functions you can quickly move from *z*-domain or *s*-domain rational system function coefficients to visualization of the filter frequency response * Magnitude * Magnitude in dB * Phase in radians * Group delay in samples or seconds (digital filter) * Group delay in seconds (analog filter)

```
[46]: def freqz_resp(b,a=[1],mode = 'dB',fs=1.0,Npts = 1024,fsize=(6,4)):
          """
          A method for displaying digital filter frequency response magnitude,
          phase, and group delay. A plot is produced using matplotlib

          freq_resp(self,mode = 'dB',Npts = 1024)

          A method for displaying the filter frequency response magnitude,
          phase, and group delay. A plot is produced using matplotlib
```

```
freqs_resp(b,a=[1],Dmin=1,Dmax=5,mode = 'dB',Npts = 1024,fsize=(6,4))

    b = ndarray of numerator coefficients
    a = ndarray of denominator coefficents
 Dmin = start frequency as 10**Dmin
 Dmax = stop frequency as 10**Dmax
 mode = display mode: 'dB' magnitude, 'phase' in radians, or
        'groupdelay_s' in samples and 'groupdelay_t' in sec,
        all versus frequency in Hz
 Npts = number of points to plot; defult is 1024
fsize = figure size; defult is (6,4) inches

Mark Wickert, January 2015
"""
f = np.arange(0,Npts)/(2.0*Npts)
w,H = signal.freqz(b,a,2*np.pi*f)
plt.figure(figsize=fsize)
if mode.lower() == 'db':
    plt.plot(f*fs,20*np.log10(np.abs(H)))
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Gain (dB)')
    plt.title('Frequency Response - Magnitude')

elif mode.lower() == 'phase':
    plt.plot(f*fs,np.angle(H))
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Phase (rad)')
    plt.title('Frequency Response - Phase')

elif (mode.lower() == 'groupdelay_s') or (mode.lower() == 'groupdelay_t'):
    """
    Notes
    -----

    Since this calculation involves finding the derivative of the
    phase response, care must be taken at phase wrapping points
    and when the phase jumps by +/-pi, which occurs when the
    amplitude response changes sign. Since the amplitude response
    is zero when the sign changes, the jumps do not alter the group
    delay results.
    """
    theta = np.unwrap(np.angle(H))
    # Since theta for an FIR filter is likely to have many pi phase
    # jumps too, we unwrap a second time 2*theta and divide by 2
    theta2 = np.unwrap(2*theta)/2.
    theta_dif = np.diff(theta2)
    f_diff = np.diff(f)
    Tg = -np.diff(theta2)/np.diff(w)
    max_Tg = np.max(Tg)
    #print(max_Tg)
    if mode.lower() == 'groupdelay_t':
        max_Tg /= fs
```

```
            plt.plot(f[:-1]*fs,Tg/fs)
            plt.ylim([0,1.2*max_Tg])
        else:
            plt.plot(f[:-1]*fs,Tg)
            plt.ylim([0,1.2*max_Tg])
        plt.xlabel('Frequency (Hz)')
        if mode.lower() == 'groupdelay_t':
            plt.ylabel('Group Delay (s)')
        else:
            plt.ylabel('Group Delay (samples)')
        plt.title('Frequency Response - Group Delay')
    else:
        s1 = 'Error, mode must be "dB", "phase, '
        s2 = '"groupdelay_s", or "groupdelay_t"'
        print(s1 + s2)
```

```
[47]: def freqs_resp(b,a=[1],Dmin=1,Dmax=5,mode = 'dB',Npts = 1024,fsize=(6,4)):
          """
          A method for displaying analog filter frequency response magnitude,
          phase, and group delay. A plot is produced using matplotlib

          freqs_resp(b,a=[1],Dmin=1,Dmax=5,mode='dB',Npts=1024,fsize=(6,4))

              b = ndarray of numerator coefficients
              a = ndarray of denominator coefficents
           Dmin = start frequency as 10**Dmin
           Dmax = stop frequency as 10**Dmax
           mode = display mode: 'dB' magnitude, 'phase' in radians, or
                  'groupdelay', all versus log frequency in Hz
           Npts = number of points to plot; defult is 1024
          fsize = figure size; defult is (6,4) inches

          Mark Wickert, January 2015
          """
          f = np.logspace(Dmin,Dmax,Npts)
          w,H = signal.freqs(b,a,2*np.pi*f)
          plt.figure(figsize=fsize)
          if mode.lower() == 'db':
              plt.semilogx(f,20*np.log10(np.abs(H)))
              plt.xlabel('Frequency (Hz)')
              plt.ylabel('Gain (dB)')
              plt.title('Frequency Response - Magnitude')

          elif mode.lower() == 'phase':
              plt.semilogx(f,np.angle(H))
              plt.xlabel('Frequency (Hz)')
              plt.ylabel('Phase (rad)')
              plt.title('Frequency Response - Phase')

          elif mode.lower() == 'groupdelay':
              """
              Notes
```

```
        -----

        See freqz_resp() for calculation details.
        """
        theta = np.unwrap(np.angle(H))
        # Since theta for an FIR filter is likely to have many pi phase
        # jumps too, we unwrap a second time 2*theta and divide by 2
        theta2 = np.unwrap(2*theta)/2.
        theta_dif = np.diff(theta2)
        f_diff = np.diff(f)
        Tg = -np.diff(theta2)/np.diff(w)
        max_Tg = np.max(Tg)
        #print(max_Tg)
        plt.semilogx(f[:-1],Tg)
        plt.ylim([0,1.2*max_Tg])
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Group Delay (s)')
        plt.title('Frequency Response - Group Delay')
    else:
        print('Error, mode must be "dB" or "phase or "groupdelay"')
```

Example: Discrete-Time Chebyshev Type I Bandpass Filter

```
[48]: import sk_dsp_comm.iir_design_helper as iird
      import sk_dsp_comm.fir_design_helper as fird
```

```
[49]: b1,a1,sos1 = iird.IIR_bpf(200,250,300,350,0.1,60.0,1000,'butter')
      b2,a2,sos2 = iird.IIR_bpf(200,250,300,350,0.1,60.0,1000,'cheby1')
```

```
[50]: figure()
      iird.freqz_resp_cas_list([sos1,sos2],'dB',1000)
      ylim([-70,0])
      grid();
      figure()
      iird.freqz_resp_cas_list([sos1,sos2],'groupdelay_t',1000)
      grid();
      figure()
      iird.sos_zplane(sos2)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/scikit-dsp-comm/envs/stable/lib/python3.
↪7/site-packages/scikit_dsp_comm-2.0.1-py3.7.egg/sk_dsp_comm/iir_design_helper.py:350:␣
↪RuntimeWarning: divide by zero encountered in log10
  plt.plot(f*fs,20*np.log10(np.abs(H)))
/home/docs/checkouts/readthedocs.org/user_builds/scikit-dsp-comm/envs/stable/lib/python3.
↪7/site-packages/scikit_dsp_comm-2.0.1-py3.7.egg/sk_dsp_comm/iir_design_helper.py:350:␣
↪RuntimeWarning: divide by zero encountered in log10
  plt.plot(f*fs,20*np.log10(np.abs(H)))
/home/docs/checkouts/readthedocs.org/user_builds/scikit-dsp-comm/envs/stable/lib/python3.
↪7/site-packages/scikit_dsp_comm-2.0.1-py3.7.egg/sk_dsp_comm/iir_design_helper.py:383:␣
↪RuntimeWarning: divide by zero encountered in log10
```

```
    idx = np.nonzero(np.ravel(20*np.log10(H[:-1]) < -400))[0]
/home/docs/checkouts/readthedocs.org/user_builds/scikit-dsp-comm/envs/stable/lib/python3.
→7/site-packages/scikit_dsp_comm-2.0.1-py3.7.egg/sk_dsp_comm/iir_design_helper.py:383:␣
→RuntimeWarning: invalid value encountered in multiply
    idx = np.nonzero(np.ravel(20*np.log10(H[:-1]) < -400))[0]
/home/docs/checkouts/readthedocs.org/user_builds/scikit-dsp-comm/envs/stable/lib/python3.
→7/site-packages/scikit_dsp_comm-2.0.1-py3.7.egg/sk_dsp_comm/iir_design_helper.py:383:␣
→RuntimeWarning: invalid value encountered in less
    idx = np.nonzero(np.ravel(20*np.log10(H[:-1]) < -400))[0]
/home/docs/checkouts/readthedocs.org/user_builds/scikit-dsp-comm/envs/stable/lib/python3.
→7/site-packages/scikit_dsp_comm-2.0.1-py3.7.egg/sk_dsp_comm/iir_design_helper.py:383:␣
→RuntimeWarning: divide by zero encountered in log10
    idx = np.nonzero(np.ravel(20*np.log10(H[:-1]) < -400))[0]
/home/docs/checkouts/readthedocs.org/user_builds/scikit-dsp-comm/envs/stable/lib/python3.
→7/site-packages/scikit_dsp_comm-2.0.1-py3.7.egg/sk_dsp_comm/iir_design_helper.py:383:␣
→RuntimeWarning: invalid value encountered in multiply
    idx = np.nonzero(np.ravel(20*np.log10(H[:-1]) < -400))[0]
/home/docs/checkouts/readthedocs.org/user_builds/scikit-dsp-comm/envs/stable/lib/python3.
→7/site-packages/scikit_dsp_comm-2.0.1-py3.7.egg/sk_dsp_comm/iir_design_helper.py:383:␣
→RuntimeWarning: invalid value encountered in less
    idx = np.nonzero(np.ravel(20*np.log10(H[:-1]) < -400))[0]
```

```
[50]: (12, 12)
```

```
<Figure size 432x288 with 0 Axes>
```

```
<Figure size 432x288 with 0 Axes>
```

```
<Figure size 432x288 with 0 Axes>
```

```
[51]: b,a = signal.cheby1(5,.1,2*array([250,300])/1000,btype='bandpass')
```

```
[52]: freqz_resp(b,a,mode='dB',fs=1000,fsize=(6,2))
grid()
ylim([-80,5]);
xlim([100,400]);
freqz_resp(b,a,mode='groupdelay_s',fs=1000,fsize=(6,2))
grid()
xlim([100,400]);
```

Example: Continuous-Time Bessel Bandpass Filter

```
[53]: bc,ac = signal.bessel(7,2*pi*array([10.0,50.0])*1e6,btype='bandpass',analog=True)
```

```
[54]: freqs_resp(bc,ac,6,9,mode='dB',fsize=(6,2))
grid()
ylim([-80,5]);
freqs_resp(bc,ac,6,9,mode='groupdelay',fsize=(6,2))
grid()
```

Second-Order Butterworth Lowpass Response

Consider a 3rd-order analog Butterworth is the s-domain having transfer function $H(s)$. Using the `scipy.signal` function `butter()` we find the coefficients to the rational transfer function of the form:

$$H(s) = \frac{\sum_{n=0}^{M} b_n s^n}{\sum_{n=0}^{N} a_n s^n} \tag{2.37}$$

```
[55]: b3,a3 = signal.butter(3,2*pi*1,analog=True)
      freqs_resp(b3,a3,-1,2,mode='dB',fsize=(6,2))
      grid()
      ylim([-80,5]);
      freqs_resp(b3,a3,-1,2,mode='groupdelay',fsize=(6,2))
      grid()
```

Obtaining the Step Response via Simulation

Time domain simulation of continuous time system can be performed using the `signal.lsim()` function. You have to make sure the time step is sufficiently small relative to the filter bandwidth.

```
[56]: t = arange(0,2,.0001)
      xs = ss.step(t)
      tout,ys,x_state = signal.lsim((b3,a3),xs,t)
      plot(t,ys)
      title(r'Third-Order Butterworth Step Response for $f_3 = 1$ Hz')
      ylabel(r'Ste Response')
      xlabel(r'Time (s)')
      grid();
```

```
[1]: %pylab inline
     import sk_dsp_comm.sigsys as ss
     import sk_dsp_comm.fir_design_helper as fir_d
     import sk_dsp_comm.iir_design_helper as iir_d
     import sk_dsp_comm.multirate_helper as mrh
     import scipy.signal as signal
     from IPython.display import Audio, display
     from IPython.display import Image, SVG
```

Populating the interactive namespace from numpy and matplotlib

```
[2]: %config InlineBackend.figure_formats=['svg'] # SVG inline viewing
```

2.1.11 Filter Design Using the Helper Modules

The Scipy package *signal* assists with the design of many digital filter types. As an alternative, here we explore the use of the filter design modules found in `scikit-dsp-comm` (https://github.com/mwickert/scikit-dsp-comm).

In this note we briefly explore the use of `sk_dsp_comm.fir_design_helper` and `sk_dsp_comm.iir_design_helper`. In the examples that follow we assume the import of these modules is made as follows:

```python
import sk_dsp_comm.fir_design_helper as fir_d
import sk_dsp_comm.iir_design_helper as iir_d
```

The functions in these modules provide an easier and more consistent interface for both finte impulse response (FIR) (linear phase) and infinite impulse response (IIR) classical designs. Functions inside these modules *wrap* `scipy.signal` functions and also incorporate new functionality.

2.1.12 Design From Amplitude Response Requirements

With both `fir_design_helper` and `iir_design_helper` a design starts with amplitude response requirements, that is the filter passband critical frequencies, stopband critical frequencies, passband ripple, and stopband attenuation. The number of taps/coefficients (FIR case) or the filter order (IIR case) needed to meet these requirements is then determined and the filter coefficients are returned as an ndarray b for FIR, and for IIR both b and a arrays, and a second-order sections `sos` 2D array, with the rows containing the corresponding cascade of second-order sections toplogy for IIR filters.

For the FIR case we have in the z-domain

$$H_{\text{FIR}}(z) = \sum_{k=0}^{N} b_k z^{-k}$$

with ndarray b = $[b_0, b_1, \ldots, b_N]$. For the IIR case we have in the z-domain

$$H_{\text{IIR}}(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{\sum_{k=1}^{N} a_k z^{-k}} \tag{2.38}$$

$$= \prod_{k=0}^{N_s-1} \frac{b_{k0} + b_{k1}z^{-1} + b_{k2}z^{-2}}{1 + a_{k1}z^{-1} + a_{k2}z^{-2}} = \prod_{k=0}^{N_s-1} H_k(z) \tag{2.39}$$

where $N_s = \lfloor (N+1)/2 \rfloor$. For the b/a form the coefficients are arranged as

```
b = [b0, b1, ..., bM-1], the numerator filter coefficients
a = [a0, a1, ..., aN-1], the denominator filter ceofficients
```

For the `sos` form each row of the 2D `sos` array corresponds to the coefficients of $H_k(z)$, as follows:

```
SOS_mat = [[b00, b01, b02, 1, a01, a02], #biquad 0
           [b10, b11, b12, 1, a11, a12], #biquad 1
            .
            .
           [bNs-10, bNs-11, bNs-12, 1, aNs-11, aNs-12]] #biquad Ns-1
```

2.1.13 Linear Phase FIR Filter Design

The primary focus of this module is adding the ability to design linear phase FIR filters from user friendly amplitude response requirements.

Most digital filter design is motivated by the desire to approach an ideal filter. Recall an ideal filter will pass signals of a certain of frequencies and block others. For both analog and digital filters the designer can choose from a variety of approximation techniques. For digital filters the approximation techniques fall into the categories of IIR or FIR. In the design of FIR filters two popular techniques are truncating the ideal filter impulse response and applying a window, and optimum equiripple approximations Oppenheim2010. Frequency sampling based approaches are also popular, but will not be considered here, even though `scipy.signal` supports all three. Filter design generally begins with a specification of the desired frequency response. The filter frequency response may be stated in several ways, but amplitude response is the most common, e.g., state how $H_c(j\Omega)$ or $H(e^{j\omega}) = H(e^{j2\pi f/f_s})$ should behave. A completed design consists of the number of coefficients (taps) required and the coefficients themselves (double precision float or `float64` in Numpy, and `float64_t` in C). Figure 1, below, shows amplitude response requirements in terms of filter gain and critical frequencies for lowpass, highpass, bandpass, and bandstop filters. The critical frequencies are given here in terms of analog requirements in Hz. The sampling frequency is assumed to be in Hz. The passband ripple and stopband attenuation values are in dB. Note in dB terms attenuation is the negative of gain, e.g., -60 of stopband gain is equivalent to 60 dB of stopband attenuation.

```
[3]: Image('300ppi/FIR_Lowpass_Highpass_Bandpass_Bandstop@300ppi.png',width='90%')
```

[3]:
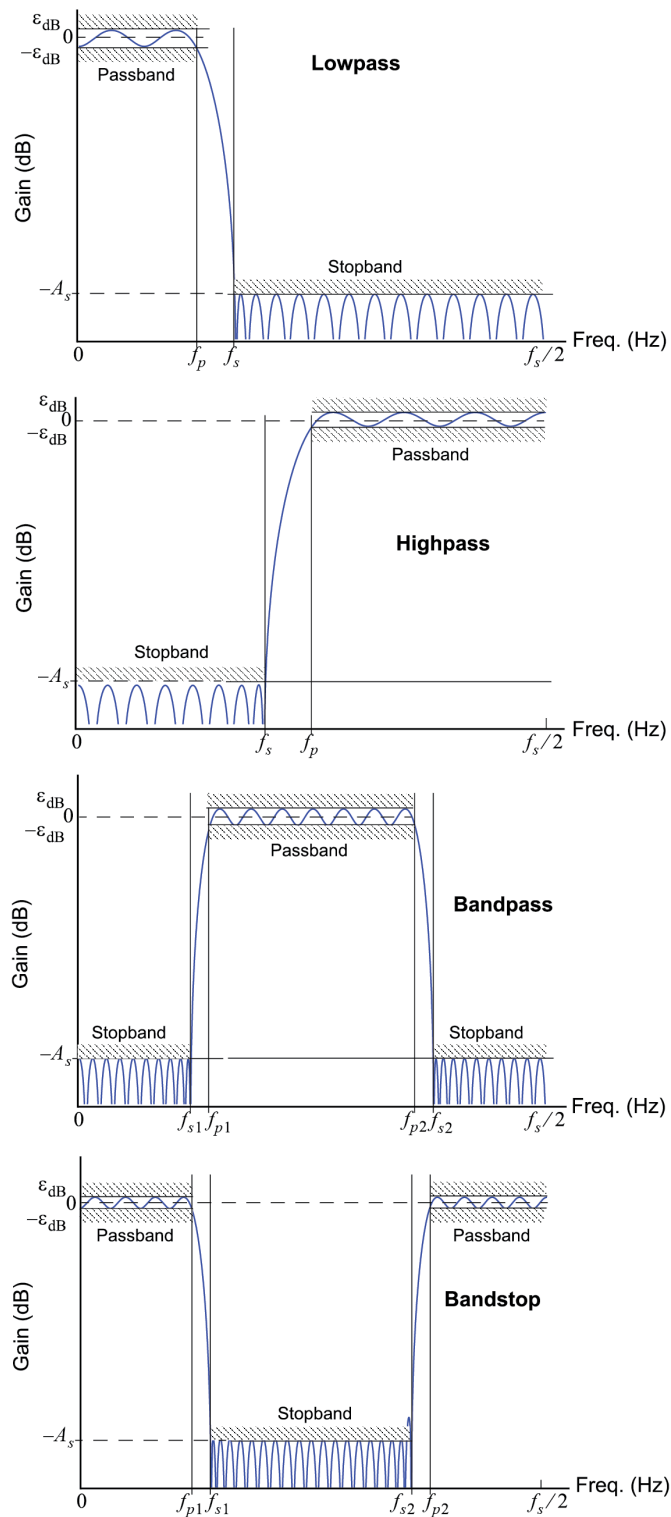
Figure: 1 General amplitude response requirements for the lowpass, highpass, bandpass, and bandstop filter types.

There are 10 filter design functions and one plotting function available in `fir_design_helper.py`. Four functions for designing Kaiser window based FIR filters and four functions for designing equiripple based FIR filters. Of the eight just described, they all take in amplitude response requirements and return a coefficients array. Two of the 10

filter functions are simply wrappers around the `scipy.signal` function `signal.firwin()` for designing filters of a specific order when one (lowpass) or two (bandpass) critical frequencies are given. The wrapper functions fix the window type to the `firwin` default of hann (hanning). The remaining eight are described below in Table 1. The plotting function provides an easy means to compare the resulting frequency response of one or more designs on a single plot. Display modes allow gain in dB, phase in radians, group delay in samples, and group delay in seconds for a given sampling rate. This function, `freq_resp_list()`, works for both FIR and IIR designs. Table 1 provides the interface details to the eight design functions where d_stop and d_pass are positive dB values and the critical frequencies have the same unit as the sampling frequency f_s. These functions do not create perfect results so some tuning of of the design parameters may be needed, in addition to bumping the filter order up or down via `N_bump`.

[4]: `Image('300ppi/FIR_Kaiser_Equiripple_Table@300ppi.png',width='80%')`

[4]:

Table 1: FIR filter design functions in `fir_design_helper.py`.

Type	FIR Filter Design Functions
Kasier Window	
Lowpass	`h_FIR = firwin_kaiser_lpf(f_pass, f_stop, d_stop, fs = 1.0, N_bump=0)`
Highpass	`h_FIR = firwin_kaiser_hpf(f_stop, f_pass, d_stop, fs = 1.0, N_bump=0)`
Bandpass	`h_FIR = firwin_kaiser_bpf(f_stop1, f_pass1, f_pass2, f_stop2, d_stop,` `fs = 1.0, N_bump=0)`
Bandstop	`h_FIR = firwin_kaiser_bsf(f_stop1, f_pass1, f_pass2, f_stop2, d_stop,` `fs = 1.0, N_bump=0)`
Equiripple Approximation	
Lowpass	`h_FIR = fir_remez_lpf(f_pass, f_stop, d_pass, d_stop, fs = 1.0,` `N_bump=5)`
Highpass	`h_FIR = fir_remez_hpf(f_stop, f_pass, d_pass, d_stop, fs = 1.0,` `N_bump=5)`
Bandpass	`h_FIR = fir_remez_bpf(f_stop1, f_pass1, f_pass2, f_stop2, d_pass,` `d_stop, fs = 1.0, N_bump=5)`
Bandstop	`h_FIR = fir_remez_bsf(f_pass1, f_stop1, f_stop2, f_pass2, d_pass,` `d_stop, fs = 1.0, N_bump=5)`
Support Function: Compare Designs	
Plot a List of Designs	`freqz_resp_list([b],a=[1],mode = 'dB',fs=1.0,Npts = 1024,fsize=(6,4))` where `[b]` is a list coefficient arrays and mode can be: `'dB'`, `'phase'` in radians, `'groupdelay_s'` in samples or `'groupdelay_t'` in seconds

The optional `N_bump` argument allows the filter order to be bumped up or down by an integer value in order to fine tune the design. Making changes to the stopband gain main also be helpful in fine tuning. Note also that the Kaiser bandstop filter order is constrained to be even (an odd number of taps).

Design Examples

Example 1: Lowpass with $f_s = 1$ Hz

For this 31 tap filter we choose the cutoff frequency to be $F_c = F_s/8$, or in normalized form $f_c = 1/8$.

[5]: `b_k = fir_d.firwin_kaiser_lpf(1/8,1/6,50,1.0)`
`b_r = fir_d.fir_remez_lpf(1/8,1/6,0.2,50,1.0)`

```
[6]: fir_d.freqz_resp_list([b_k,b_r],[[1],[1]],'dB',fs=1)
     ylim([-80,5])
     title(r'Kaiser vs Equal Ripple Lowpass')
     ylabel(r'Filter Gain (dB)')
     xlabel(r'Frequency in kHz')
     legend((r'Kaiser: %d taps' % len(b_k),r'Remez: %d taps' % len(b_r)),loc='best')
     grid();
```

```
[7]: b_k_hp = fir_d.firwin_kaiser_hpf(1/8,1/6,50,1.0)
     b_r_hp = fir_d.fir_remez_hpf(1/8,1/6,0.2,50,1.0)
```

```
[8]: fir_d.freqz_resp_list([b_k_hp,b_r_hp],[[1],[1]],'dB',fs=1)
     ylim([-80,5])
     title(r'Kaiser vs Equal Ripple Lowpass')
     ylabel(r'Filter Gain (dB)')
     xlabel(r'Frequency in kHz')
     legend((r'Kaiser: %d taps' % len(b_k),r'Remez: %d taps' % len(b_r)),loc='best')
     grid();
```

```
[9]: b_k_bp = fir_d.firwin_kaiser_bpf(7000,8000,14000,15000,50,48000)
     b_r_bp = fir_d.fir_remez_bpf(7000,8000,14000,15000,0.2,50,48000)
```

```
[10]: fir_d.freqz_resp_list([b_k_bp,b_r_bp],[[1],[1]],'dB',fs=48)
      ylim([-80,5])
      title(r'Kaiser vs Equal Ripple Bandpass')
      ylabel(r'Filter Gain (dB)')
      xlabel(r'Frequency in kHz')
      legend((r'Kaiser: %d taps' % len(b_k_bp),
              r'Remez: %d taps' % len(b_r_bp)),
            loc='lower right')
      grid();
```

A Design Example Useful for Interpolation or Decimation

Here we consider a lowpass design that needs to pass frequencies from [0, 4000] Hz with a sampling rate of 96000 Hz. This scenario arises when building an interpolator using the classes of the `scikit-dps-comm` module `multirate_helper.py` to increase the sampling rate from 8000 Hz to 96000 Hz, or an interpolation factor of $L = 12$. Note at the top of this notebook we have also have the import

```
import sk_dsp_comm.multirate_helper as mrh
```

so that some of the functionality can be accessed. For more details on the use of `multirate_helper` see.

Start with an equalripple design having transition band centered on 4000 Hz with passband ripple of 0.5 dB and stopband attenuation of 60 dB.

```
[11]: b_up = fir_d.fir_remez_lpf(3300,4300,0.5,60,96000)
```

```
[12]: mr_up = mrh.multirate_FIR(b_up)
```

- Consider the pole-zero configuration for this high-order filter

```
[13]: # Take a look at the pole-zero configuration of this very
      # high-order (many taps) linear phase FIR
      mr_up.zplane()
```

- Check out the passband and stopband gains

```
[14]: # Verify the passband and stopband gains are as expected
      mr_up.freq_resp('db',96000)
```

- See that the group delay is the expected value of $(N_{\text{taps}} - 1)/2 = 98$ samples

```
[15]: (len(b_up-1))/2
```

```
[15]: 98.0
```

```
[16]: # Verify that the FIR design has constant group delay (N_taps - 1)/2 samples
      mr_up.freq_resp('groupdelay_s',96000,[0,100])
```

The object `mr_up` can now be used for interpolation or decimation with a rate change factor of 12.

2.1.14 Traditional IIR Filter Design using the Bilinear Transform

The scipy.signal package fully supports the design of IIR digital filters from analog prototypes. IIR filters like FIR filters, are typically designed with amplitude response requirements in mind. A collection of design functions are available directly from `scipy.signal` for this purpose, in particular the function `scipy.signal.iirdesign()`. To make the design of lowpass, highpass, bandpass, and bandstop filters consistent with the module `fir_design_helper.py` the module `iir_design_helper.py` was written. Figure 2, below, details how the amplitude response parameters are defined graphically.

```
[17]: Image('300ppi/IIR_Lowpass_Highpass_Bandpass_Bandstop@300ppi.png',width='90%')
```

[17]:
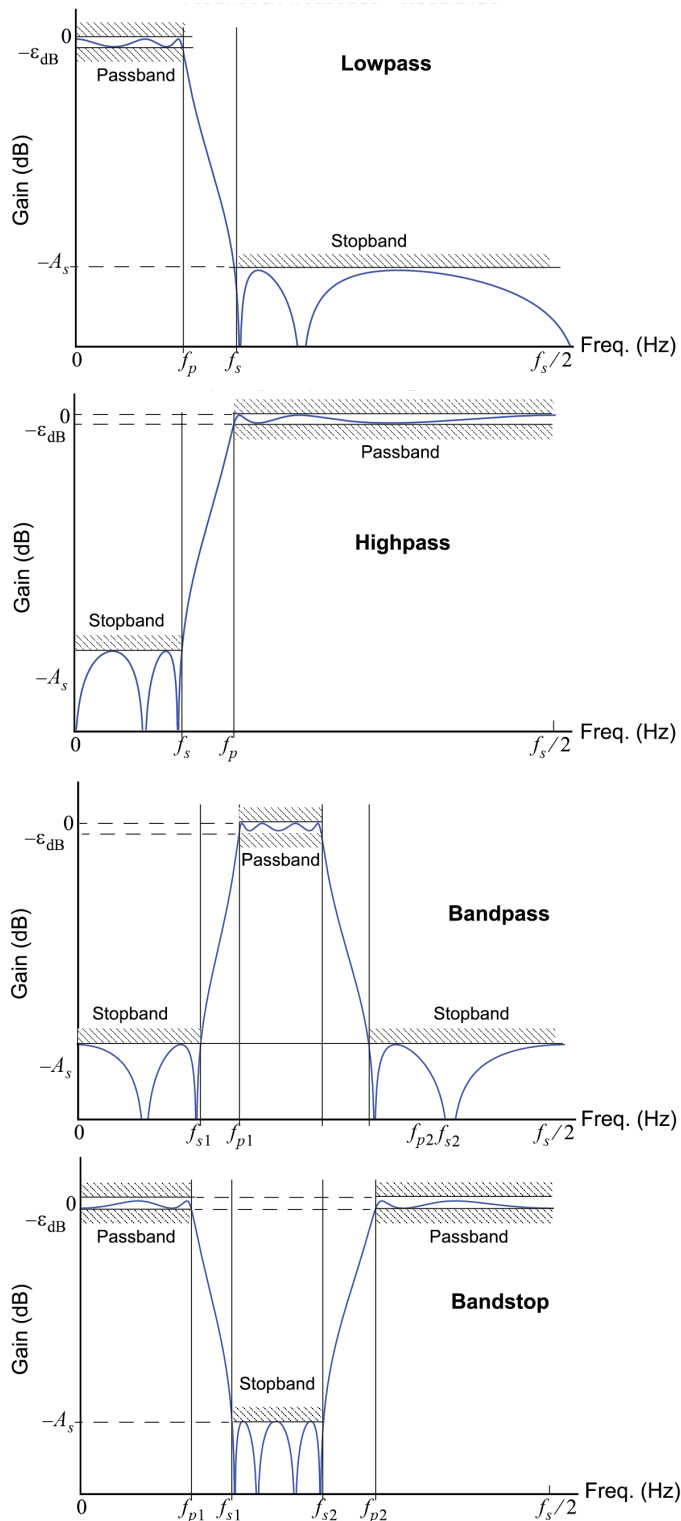
Figure 2: General amplitude response requirements for the lowpass, highpass, bandpass, and bandstop IIR filter types.

Within `iir_design_helper.py` there are four filter design functions and a collection of supporting functions available. The four filter design functions are used for designing lowpass, highpass, bandpass, and bandstop filters, utilizing Butterworth, Chebshev type 1, Chebyshev type 2, and elliptical filter prototypes. See Oppenheim2010 and ECE 5650

notes Chapter 9 for detailed design information. The function interfaces are described in Table 2.

```
[18]: Image('300ppi/IIR_Table@300ppi.png',width='80%')
```

[18]:

Table 2: IIR filter design functions in `iir_design_helper.py` and key support functions.

Type	IIR Filter Design Functions[*]
Transfer Function (b,a) and SOS	

Lowpass (bilinear)	`b, a, sos = IIR_lpf(f_pass, f_stop, Ripple_pass, Atten_stop,`
	`                     fs = 1.00, ftype = 'butter')`
	ftype may be `'butter'`, `'butter'cheby1'`, `'cheby2'`, or `'elliptic'`

Highpass (bilinear)	`b, a, sos = IIR_hpf(f_stop, f_pass, Ripple_pass, Atten_stop,`
	`                     fs = 1.00, ftype = 'butter')`
	ftype may be `'butter'`, `'butter'cheby1'`, `'cheby2'`, or `'elliptic'`

Bandpass (bilinear)	`b, a, sos = IIR_bpf(f_stop1, f_pass1, f_pass2, f_stop2, Ripple_pass,`
	`                     Atten_stop, fs = 1.00, ftype = 'butter')`
	ftype may be `'butter'`, `'butter'cheby1'`, `'cheby2'`, or `'elliptic'`

Bandstop (bilinear)	`b, a, sos = IIR_bsf(f_pass1, f_stop1, f_stop2, f_pass2, Ripple_pass,`
	`                     Atten_stop, fs = 1.00, ftype = 'butter')`
	ftype may be `'butter'`, `'butter'cheby1'`, `'cheby2'`, or `'elliptic'`

Support Functions	
SOS list plot	`freqz_resp_cas_list([sos],mode = 'dB',fs=1.0,Npts = 1024,fsize=(6,4))` where: [sos] is a list coefficient arrays and mode can be: `'dB'`, `'phase'` in radians, `'groupdelay_s'` in samples or `'groupdelay_t'` in seconds
SOS `freqz`	`w, Hcas = freqz_cas(sos,w)` freqz for a single sos section
SOS plot pole-zero	`sos_zplane(sos,auto_scale=True,size=2,tol = 0.001)` More accurate root factoring results in a more accurate pole-zero plot.
Cascade SOS	`sos = sos_cascade(sos1,sos2)` Create a new sos by cascading sos1 with sos2

[*]These functions wrap `scipy.signal.iirdesign()` to provide an interface more consistent with the FIR design functions found in the module `fir_design_helper.py`. The function `unique_cpx_roots()` is used to mark repeated poles and zeros in `sos_zplane`. Note: All critical frequencies given in increasing order.

The filter functions return the filter coefficients in two formats:

1. Traditional transfer function form as numerator coefficients `b` and denominator `a` coefficients arrays, and

2. Cascade of biquadratic sections form using the previously introduced sos 2D array or matrix.

Both are provided to allow further analysis with either a direct form topology or the sos form. The underlying `signal.iirdesign()` function also provides a third option: a list of poles and zeros. The `sos` form desireable for high precision filters, as it is more robust to coefficient quantization, in spite using double precision coefficients in the `b` and `a` arrays.

Of the remaining support functions four are also described in Table 2, above. The most significant functions are `freqz_resp_cas_list`, available for graphically comparing the frequency response over several designs, and `sos_zplane` a function for plotting the pole-zero pattern. Both operate using the `sos` matrix. A transfer function form (`b/a`) for frequency response plotting, `freqz_resp_list`, is also present in the module. This function was first introduced in the FIR design section. The frequency response function plotting offers modes for gain in dB, phase in radians, group delay in samples, and group delay in seconds, all for a given sampling rate in Hz. The pole-zero plotting function locates pole and zeros more accurately than `sk_dsp_commsigsys.zplane`, as the numpy function `roots()` is only solving quadratic polynomials. Also, repeated roots can be displayed as theoretically expected, and also so noted in the graphical display by superscripts next to the pole and zero markers.

IIR Design Based on the Bilinear Transformation

There are multiple ways of designing IIR filters based on amplitude response requirements. When the desire is to have the filter approximation follow an analog prototype such as Butterworth, Chebychev, etc., is using the bilinear transformation. The function `signal.iirdesign()` described above does exactly this.

In the example below we consider lowpass amplitude response requirements and see how the filter order changes when we choose different analog prototypes.

Example: Lowpass Design Comparison

The lowpass amplitude response requirements given $f_s = 48$ kHz are: 1. $f_{pass} = 5$ kHz 2. $f_{stop} = 8$ kHz 3. Passband ripple of 0.5 dB 4. Stopband attenuation of 60 dB

Design four filters to meet the same requirements: `butter`, `cheby1`, ,`cheby2`, and `ellip`:

```
[19]: fs = 48000
      f_pass = 5000
      f_stop = 8000
      b_but,a_but,sos_but = iir_d.IIR_lpf(f_pass,f_stop,0.5,60,fs,'butter')
      b_cheb1,a_cheb1,sos_cheb1 = iir_d.IIR_lpf(f_pass,f_stop,0.5,60,fs,'cheby1')
      b_cheb2,a_cheb2,sos_cheb2 = iir_d.IIR_lpf(f_pass,f_stop,0.5,60,fs,'cheby2')
      b_elli,a_elli,sos_elli = iir_d.IIR_lpf(f_pass,f_stop,0.5,60,fs,'ellip')
```

Frequency Response Comparison

Here we compare the magnitude response in dB using the `sos` form of each filter as the input. The elliptic is the most efficient, and actually over achieves by reaching the stopband requirement at less than 8 kHz.

```
[20]: iir_d.freqz_resp_cas_list([sos_but,sos_cheb1,sos_cheb2,sos_elli],'dB',fs=48)
      ylim([-80,5])
      title(r'IIR Lowpass Compare')
      ylabel(r'Filter Gain (dB)')
      xlabel(r'Frequency in kHz')
      legend((r'Butter order: %d' % (len(a_but)-1),
              r'Cheby1 order: %d' % (len(a_cheb1)-1),
              r'Cheby2 order: %d' % (len(a_cheb2)-1),
              r'Elliptic order: %d' % (len(a_elli)-1)),loc='best')
      grid();
```

Next plot the pole-zero configuration of just the butterworth design. Here we use the a special version of `ss.zplane` that works with the `sos` 2D array.

```
[21]: iir_d.sos_zplane(sos_but)
```

```
[21]: (15, 15)
```

Note the two plots above can also be obtained using the transfer function form via `iir_d.freqz_resp_list([b], [a],'dB',fs=48)` and `ss.zplane(b,a)`, respectively. The `sos` form will yield more accurate results, as it is less sensitive to coefficient quantization. This is particularly true for the pole-zero plot, as rooting a 15th degree polynomial is far more subject to errors than rooting a simple quadratic.

For the 15th-order Butterworth the bilinear transformation maps the expected 15 s-domain zeros at infinity to $z = -1$. If you use `sk_dsp_comm.sigsys.zplane()` you will find that the 15 zeros at are in a tight circle around $z = -1$, indicating polynomial rooting errors. Likewise the frequency response will be more accurate.

Signal filtering of ndarray `x` is done using the filter designs is done using functions from `scipy.signal`:

1. For transfer function form `y = signal.lfilter(b,a,x)`

2. For sos form `y = signal.sosfilt(sos,x)`

A Half-Band Filter Design to Pass up to $W/2$ when $f_s = 8$ kHz

Here we consider a lowpass design that needs to pass frequencies up to $f_s/4$. Specifically when $f_s = 8000$ Hz, the filter passband becomes [0, 2000] Hz. Once the coefficients are found a `mrh.multirate` object is created to allow further study of the filter, and ultimately implement filtering of a white noise signal.

Start with an elliptical design having transition band centered on 2000 Hz with passband ripple of 0.5 dB and stopband attenuation of 80 dB. The transition bandwidth is set to 100 Hz, with 50 Hz on either side of 2000 Hz.

```
[22]: # Elliptic IIR Lowpass
      b_lp,a_lp,sos_lp = iir_d.IIR_lpf(1950,2050,0.5,80,8000.,'ellip')
      mr_lp = mrh.multirate_IIR(sos_lp)
```

```
[23]: mr_lp.freq_resp('db',8000)
```

Pass Gaussian white noise of variance $\sigma_x^2 = 1$ through the filter. Use a lot of samples so the spectral estimate can accurately form $S_y(f) = \sigma_x^2 \cdot |H(e^{j2\pi f/f_s})|^2 = |H(e^{j2\pi f/f_s})|^2$.

```
[24]: x = randn(1000000)
      y = mr_lp.filter(x)
      psd(x,2**10,8000);
      psd(y,2**10,8000);
      title(r'Filtering White Noise Having $\sigma_x^2 = 1$')
      legend(('Input PSD','Output PSD'),loc='best')
      ylim([-130,-30])
```

```
[24]: (-130.0, -30.0)
```

```
[25]: fs = 8000
      print('Expected PSD of %2.3f dB/Hz' % (0-10*log10(fs),))
```

```
Expected PSD of -39.031 dB/Hz
```

Amplitude Response Bandpass Design

Here we consider FIR and IIR bandpass designs for use in an SSB demodulator to remove potential adjacent channel signals sitting either side of a frequency band running from 23 kHz to 24 kHz.

```
[26]: b_rec_bpf1 = fir_d.fir_remez_bpf(23000,24000,28000,29000,0.5,70,96000,8)
      fir_d.freqz_resp_list([b_rec_bpf1],[1],mode='dB',fs=96000)
      ylim([-80, 5])
      grid();
```

The group delay is flat (constant) by virture of the design having linear phase.

```
[27]: b_rec_bpf1 = fir_d.fir_remez_bpf(23000,24000,28000,29000,0.5,70,96000,8)
      fir_d.freqz_resp_list([b_rec_bpf1],[1],mode='groupdelay_s',fs=96000)
      grid();
```

Compare the FIR design with an elliptical design:

```
[28]: b_rec_bpf2,a_rec_bpf2,sos_rec_bpf2 = iir_d.IIR_bpf(23000,24000,28000,29000,
                                                         0.5,70,96000,'ellip')
      with np.errstate(divide='ignore'):
          iir_d.freqz_resp_cas_list([sos_rec_bpf2],mode='dB',fs=96000)
      ylim([-80, 5])
      grid();
```

This high order elliptic has a nice tight amplitude response for minimal coefficients, but the group delay is terrible:

```
[29]: with np.errstate(divide='ignore', invalid='ignore'): #manage singularity warnings
          iir_d.freqz_resp_cas_list([sos_rec_bpf2],mode='groupdelay_s',fs=96000)
      #ylim([-80, 5])
      grid();
```

```
[1]: %pylab inline
     import sk_dsp_comm.sigsys as ss
     import sk_dsp_comm.fir_design_helper as fir_d
     import sk_dsp_comm.iir_design_helper as iir_d
     import sk_dsp_comm.multirate_helper as mrh
     import scipy.signal as signal
     from IPython.display import Audio, display
     from IPython.display import Image, SVG
```

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

```
[2]: %config InlineBackend.figure_formats=['svg'] # SVG inline viewing
```

2.1.15 Multirate Signal Processing Using `multirate_helper`

In this section the classes `multirate_FIR` and `multirate_IIR`, found in the module `sk_dsp_comm.multirate_helper`, are discussed with the aim of seeing how they can be used to filter, interpolate (upsample and filter), and decimate (filter and downsample) discrete time signals. Fundamentally the processing consists of two elements: (1) and upsampler or downsampler and (2) a lowpass filter.

Fundamentally this modules provides classes to change the sampling rate by an integer factor, either up, *interpolation* or down, *decimation*, with integrated filtering to supress spectral images or aliases, respectively. The top level block diagram of the interpolator and decimator are given in the following two figures. The frequencies given in the figures assume that the interpolator is rate chainging from 8 ksps to 96 ksps ($L = 12$) and the decimator is rate changing from 96 ksps to 8 ksps ($M = 12$). This is for example purposes only. The FIR/IIR filter cutoff frequency will in general be $f_c = f_{s,\text{out}}/(2L)$ for the decimator and $f_c = f_{s,\text{in}}/(2M)$. The primitives to implement the classes are available in `sk_dsp_comm.sigsys` and `scipy.signal`.

```
[3]: Image('300ppi/Interpolator_Top_Level@300ppi.png',width='60%')
```

[3]:
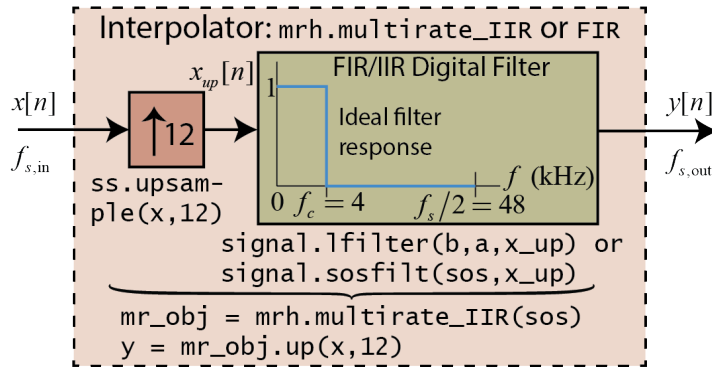

```
[4]: Image('300ppi/Decimator_Top_Level@300ppi.png',width='60%')
```

[4]:
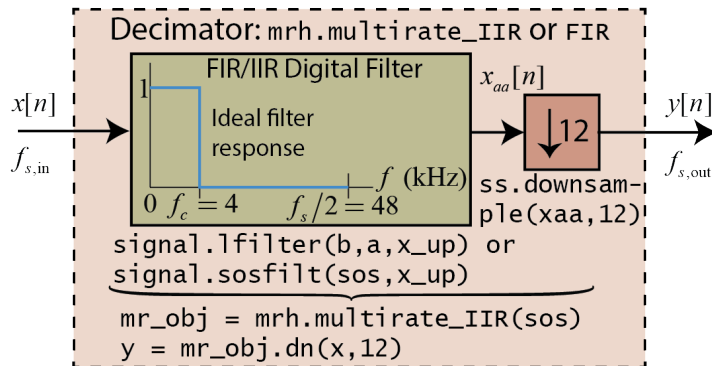
The upsample block, shown above with arrow pointing up and integer $L = 12$ next to the arrow, takes the input sequence and produces the output sequence by inserting $L - 1$ (as shown here 11) zero samples between each input sample. The downsample block, shown above with arrow pointing down and integer $M = 12$ next to the arrow, takes the input sequence and retains at the output sequence every Mth (as shown here 12th) sample.

The impact of these blocks in the frequency domain is a little harder to explain. In words, the spectrum at the output of the upsampler is compressed by the factor L, such that it will contain L spectral images, including the fundamental image centered at $f = 0$, evenly spaced up to the sampling f_s. Overall the spectrum of $x_{up}[n]$ is of course periodic with respect to the sampling rate. The lowpass filter interpolates signal sample values from the non-zero samples where the zero samples reside. It is this interpolation that effectively removed or suppresses the spectral images outside the interval $|f| > f_s/(2L)$.

For the downsampler the input spectrum is stretched along the frequency axis by the factor M, with aliasing from frequency bands outside $|f| < f_s/(2M)$. To avoid aliasing the lowpass filter blocks input signals for $f > f_s/(2M)$.

To get started using the module you will need an `import` similar to:

```
import sk_dsp_comm.multirate_helper as mrh
```
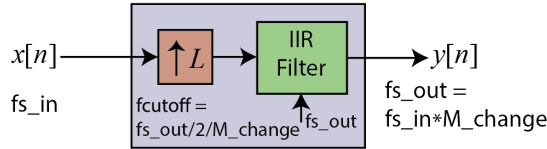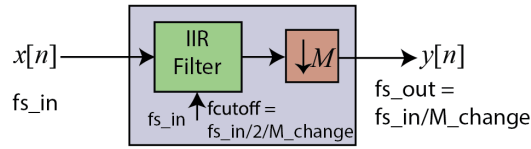
The `rate_change` Class

We start with the description of a third class, `mrh.rate_change`, which is simplistic, offering little user interaction, but automatically designs the required lowpass filter you see in the above block diagrams. Below is a table which describes this class:

```
[5]: Image('300ppi/Multirate_Table1@300ppi.png',width='85%')
```

[5]:

Table 1: Classes and functions in `multirate_helper.py`.

Type	Class/Method/Function
module class: multirate_helper.rate_change	

constructor	`rc_obj = rate_change(M_change = 12,fcutoff=0.9,N_filt_order=8,`
	`                          ftype='butter')`
	`fcutoff` is normalized to the sampling rate, e.g., `fs_out/M_change` for interpolation and `fs_in/M_change` for decimation.
	`ftype` may also be `'cheby1'` where the ripple is fixed at 0.05 dB.

interpolate	`y = rc_obj.up(x)`
decimate	`y = rc_obj.dn(x)`
Notes:	

This class is used in the analog modulation demos for the ECE 4625/5625 Chapter 3 Jupyter notebook. Using this class you can quickly create a interpolation or decimation block with the necessary lowpass filter automatically designed and implemented. Fine tuning of the filter is limited to choosing the filter order and the cutoff frequency as a fraction of the signal bandwidth given the rate change integer, L or M. The filter type is also limited to Butterworth or Chebyshev type 1 having passband ripple of 0.05 dB.

A Simple Example

Pass a sinusoidal signal through an $L = 4$ interpolator. Verify that spectral images occur with the use of the interpolation lowpass filter.

```
[6]: fs_in = 8000
M = 4
fs_out = M*fs_in
rc1 = mrh.rate_change(M) # Rate change by 4
n = arange(0,1000)
x = cos(2*pi*1000/fs_in*n)
x_up = ss.upsample(x,4)
y = rc1.up(x)
```

Time Domain

```
[7]: subplot(211)
     stem(n[500:550],x_up[500:550]);
     ylabel(r'$x_{up}[n]$')
     title(r'Upsample by $L=4$ Output')
     #ylim(-100,-10)
     subplot(212)
     stem(n[500:550],y[500:550]);
     ylabel(r'$y[n]$')
     xlabel(r'')
     title(r'Interpolate by $L=4$ Output')
     #ylim(-100,-10)
     tight_layout()
```

- Clearly the lowpass interpolation filter has done a good job of filling in values for the zero samples

Frequency Domain

```
[8]: subplot(211)
     psd(x_up,2**10,fs_out);
     ylabel(r'PSD (dB)')
     title(r'Upsample by $L=4$ Output')
     ylim(-100,-10)
     subplot(212)
     psd(y,2**10,fs_out);
     ylabel(r'PSD (dB)')
     title(r'Interpolate by $L=4$ Output')
     ylim(-100,-10)
     tight_layout()
```

- The filtering action of the LPF does its best to suppress the images at 7000, 9000, and 15000 Hz.

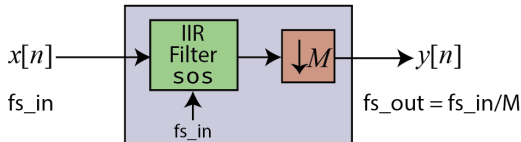**The `multirate_FIR` Class**
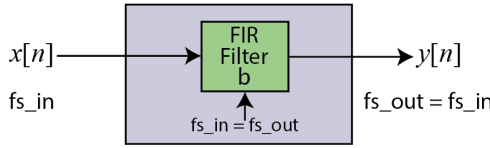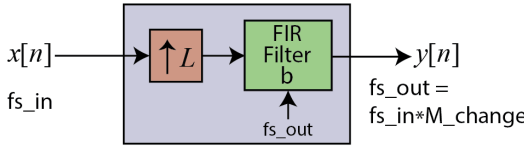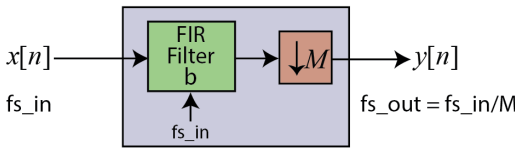
With this class you implement an object that can filter, interpolate, or decimate a signal. Additionally support methods drill into the characteristics of the lowpass filter at the heart of the processing block. To use this class the user must supply FIR filter coefficients that implement a lowpass filter with cutoff frequency appropriate for the desired interpolation of decimation factor. The module `sk_dsp_com.FIR_design_helper` is capable of delivering the need filter coefficients array. See FIR design helper notes for multirate filter design examples.

With FIR coefficients in hand it is an easy matter to create an multirate FIR object capable of filtering, interpolation, or decimation. The details of the class interface are given in Table 2 below.

```
[9]: Image('300ppi/Multirate_Table2@300ppi.png',width='85%')
```

[9]:

Table 2: Classes and functions in `multirate_helper.py` (cont.).

Type	Class/Method/Function
module class: multirate_helper.multirate_FIR	
constructor	`mr_obj = multirate_FIR(b)` b : user supplied FIR filter that meets the desired filtering, interpolation, or decimation needs; nominally design filter for `fc = (fs/2)/M_change`
filter	`y = mr_obj.filter(x)`
interpolate	`y = up(x,L_change = 12)`
decimate	`y = dn(x,M_change = 12)`
plot H	`mr_obj.freq_resp(mode = 'dB', fs = 8000, ylim = [-100,2])` mode : `'dB'` magnitude, `'phase'` in radians, or `'groupdelay_s'` in samples and `'groupdelay_t'` in sec, all versus frequency in Hz
plot PZ	`mr_obj.zplane(auto_scale=True,size=2,detect_mult=True,tol=0.001)`
function:	
plot H	`freqz_resp(b,a=[1],mode = 'dB',fs=1.0,Npts = 1024,fsize=(6,4))` mode : `'dB'` magnitude, `'phase'` in radians, or `'groupdelay_s'` in samples and `'groupdelay_t'` in sec, all versus frequency in Hz
Notes:	

Notice that the class also provides a means to obtain frequency response plots and pole-zero plots directly from the instantiated multirate objects.

FIR Interpolator Design Example

Here we take the earlier lowpass filter designed to interpolate a signal being upsampled from $f_{s1} = 8000$ kHz to $f_{s2} = 96$ kHz. The upsampling factor is $L = f_{s2}/f_{s1} = 12$. The ideal interpolation filter should cutoff at $f_{s1}/2 = f_{s2}/(2 \cdot 12) = 8000/2 = 4000$ Hz.

Recall the upsampler (`y = ss.upsampler(x, L)`) inserts $L-1$ samples between each input sample. In the frequency domain the zero insertion replicates the input spectrum on $[0, f_{s1}/2]$ L times over the interval $[0, f_{s2}]$ (equivalently $L/2$ times on the inteval $[0f_{s2}/2]$). The lowpass interpolation filter serves to removes the images above $f_{s2}/(2L)$ in the frequency domain and in so doing filling in the zeros samples with waveform interpolants in the time domain.

```
[10]: # Design the filter core for an interpolator used in changing the sampling rate from␣
      ↪8000 Hz
      # to 96000 Hz
      b_up = fir_d.fir_remez_lpf(3300,4300,0.5,60,96000)
      # Create the multirate object
      mrh_up = mrh.multirate_FIR(b_up)
```

As an input consider a sinusoid at 1 kHz and observe the interpolator output spectrum compared with the input spectrum.

```
[11]: # Sinusoidal test signal
      n = arange(10000)
      x = cos(2*pi*1000/8000*n)
      # Interpolate by 12 (upsample by 12 followed by lowpass filter)
      y = mrh_up.up(x,12)
```

```
[12]: # Plot the results
      subplot(211)
      psd(x,2**12,8000);
      title(r'1 KHz Sinusoid Input to $L=12$ Interpolator')
      ylabel(r'PSD (dB)')
      ylim([-100,0])
      subplot(212)
      psd(y,2**12,12*8000)
      title(r'1 KHz Sinusoid Output from $L=12$ Interpolator')
      ylabel(r'PSD (dB)')
      ylim([-100,0])
      tight_layout()
```

In the above spectrum plots notice that images of the input 1 kHz sinusoid are down $\simeq 60$ dB, which is precisely the stop band attenuation provided by the interpolation filter. The variation is due to the stopband ripple.
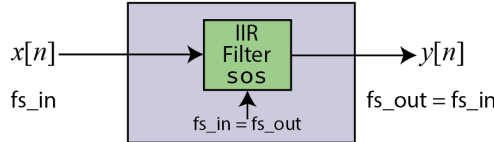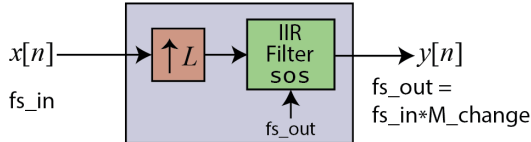
The `multirate_IIR` Class

With this class, as with `multirate_FIR` you implement an object that can filter, interpolate, or decimate a signal. The filter in this case is a user supplied IIR filter in second-order sections (`sos`) form. Additionally support methods drill into the characteristics of the lowpass filter at the heart of the procssing block. The module `sk_dsp_com.IIR_design_helper` is capable of delivering the need filter coefficients array. See IIR design helper notes for multirate filter design examples.

With IIR coefficients in hand it is an easy matter to create an multirate IIR object capable of filtering, interpolation, or decimation. The details of the class interface are given in Table 3 below.

[13]: `Image('300ppi/Multirate_Table3@300ppi.png',width='85%')`

[13]:

Table 3: Classes and functions in `multirate_helper.py` (cont.).

Type	Class/Method/Function
module class: multirate_helper.multirate_IIR	
constructor	`mr_obj = multirate_IIR(sos)` b : user supplied IIR filter that meets the desired filtering, interpolation, or decimation needs; nominally design filter for `fc = (fs/2)/M_change`
filter	`y = mr_obj.filter(x)`
interpolate	`y = up(x,L_change = 12)`
decimate	`y = dn(x,M_change = 12)`
plot H	`mr_obj.freq_resp(mode= 'dB', fs = 8000, ylim = [-100,2])` mode : `'dB'` magnitude, `'phase'` in radians, or `'groupdelay_s'` in samples and `'groupdelay_t'` in sec, all versus frequency in Hz
plot PZ	`mr_obj.zplane(auto_scale=True,size=2,detect_mult=True,tol=0.001)`
module function:	
plot H from b, a	`freqz_resp(b,a=[1],mode = 'dB',fs=1.0,Npts = 1024,fsize=(6,4))` mode : `'dB'` magnitude, `'phase'` in radians, or `'groupdelay_s'` in samples and `'groupdelay_t'` in sec, all versus frequency in Hz
Notes:	

IIR Decimator Design Example

Whan a signal is decimated the signal is first lowpass filtered then downsampled. The lowpass filter serves to prevent aliasing as the sampling rate is reduced. Downsampling by M (`y = ss.downsample(x, M)`) removes $M-1$ sampling for every M sampling input or equivalently retains one sample out of M. The lowpass prefilter has cutoff frequency equal to the folding frequency of the output sampling rate, i.e., $f_c = f_{s2}/2$. Note avoid confusion with the project requirements, where the decimator is needed to take a rate f_{s2} signal back to f_{s1}, let the input sampling rate be $f_{s2} = 96000$ HZ and the output sampling rate be $f_{s1} = 8000$ Hz. The input sampling rate is M times the output rate, i.e., $f_{s2} = M f_{s1}$, so you design the lowpass filter to have cutoff $f_c = f_{s2}/(2 \cdot L)$.

ECE 5625 Important Observation: In the coherent SSB demodulator of Project 1, the decimator can be conveniently integrated with the lowpass filter that serves to remove the double frequency term.

In the example that follows a Chebyshev type 1 lowpass filter is designed to have cutoff around 4000 Hz. A sinusoid is used as a test input signal at sampling rate 96000 Hz.

```
[14]: # Design the filter core for a decimator used in changing the
      # sampling rate from 96000 Hz to 8000 Hz
      b_dn, a_dn, sos_dn = iir_d.IIR_lpf(3300,4300,0.5,60,96000,'cheby1')
      # Create the multirate object
      mrh_dn = mrh.multirate_IIR(sos_dn)
      mrh_dn.freq_resp('dB',96000)
      title(r'Decimation Filter Frequency Response - Magnitude');
```

- Note the Chebyshev lowpass filter design above is very efficient compared with the 196-tap FIR lowpass designed for use in the interpolator. It is perhaps a better overall choice. The FIR has linear phase and the IIR filter does not, but for the project this is not really an issue.

As an input consider a sinusoid at 1 kHz and observe the interpolator output spectrum compared with the input spectrum.

```
[15]: # Sinusoidal test signal
      n = arange(100000)
      x = cos(2*pi*1000/96000*n)
      # Decimate by 12 (lowpass filter followed by downsample by 12)
      y = mrh_dn.dn(x,12)
```

```
[16]: # Plot the results
      subplot(211)
      psd(x,2**12,96000);
      title(r'1 KHz Sinusoid Input to $M=12$ Decimator')
      ylabel(r'PSD (dB)')
      ylim([-100,0])
      subplot(212)
      psd(y,2**12,8000)
      title(r'1 KHz Sinusoid Output from $M=12$ Decimator')
      ylabel(r'PSD (dB)')
      ylim([-100,0])
      tight_layout()
```

```
[1]: %pylab inline
     #%matplotlib qt
     import sk_dsp_comm.sigsys as ss
     import scipy.signal as signal
     from IPython.display import Audio, display
     from IPython.display import Image, SVG
```

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

```
[2]: 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
```

```
[3]: import scipy.special as special
     import sk_dsp_comm.digitalcom as dc
     import sk_dsp_comm.fec_conv as fec
```

Convolutional Coding

Rate 1/2

A convolutional encoder object can be created with the `fec.FECConv` method. The rate of the object will be determined by the number of generator polynomials used. Right now, only rate 1/2 and rate 1/3 are supported, so 2 or three generator polynomials can be used. The following table shows ideal rate 1/2 generator polynomials. These are also included in the docstring.

Table 1: Weight spectra :math:`c_k` for bounding the codedrate 1/2 BEP.

CL	Polynomials	D_{free}	d_f	d_f+1	d_f+2	d_f+3	d_f+4	d_f+5	d_f+6	d_f+7
3	(5,7) = ('101','111')	5	1	4	12	32	80	192	488	1024
4	(15,17) = ('1101','1111')	6	2	7	18	49	130	333	836	2069
5	(23,35) = ('10011','11101')	7	4	12	20	72	225	500	1324	3680
6	(53,75) = ('101011','111101')	8	2	36	32	62	332	701	2342	5503
7	(133,171) = ('1011011','1111001')	10	36	0	211	0	1404	0	11633	0

In addition to the generator polynomials, you can specify a decision depth for the object. This will determine how many state transitions will be used for the traceback. The following shows how to create a rate 1/2 `fec_conv` object with contraint length 3 and decision depth 10.

```
[4]: cc1 = fec.FECConv(('111','101'),10)
```

The `trellis_plot()` method can be used to see the state transitions of the `fec_conv` object.

```
[5]: cc1.trellis_plot()
```

```
/home/docs/.pyenv/versions/3.7.9/lib/python3.7/site-packages/numpy/core/_asarray.py:136:␣
→VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is␣
→a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is␣
→deprecated. If you meant to do this, you must specify 'dtype=object' when creating the␣
→ndarray
  return array(a, dtype, copy=False, order=order, subok=True)
```

Rate 1/2 Hard Decision Decoding

Now, we would like to know the theoretical bit error probability bounds of our convolutional encoding/decoding setup. We can do this using the `fec.conv_Pb_bound` method. The method takes the rate, degrees of freedom, c_k values, SNR, hard or soft decisions, and order M for an MPSK modulation scheme as arguments. It returns the BEP. The following shows theoretical bounds for rate 1/2 encoding/decoding BPSK system. Compare with Ziemer pg 667.

Weight Structure Bounds BEP

```
[6]: SNRdB = arange(0,12,.1)
     Pb_uc = fec.conv_Pb_bound(1/2,7,[4, 12, 20, 72, 225],SNRdB,2)
     Pb_s_half_3_hard = fec.conv_Pb_bound(1/2,5,[1, 4, 12, 32, 80, 192, 448, 1024],SNRdB,0)
     Pb_s_half_5_hard = fec.conv_Pb_bound(1/2,7,[4, 12, 20, 72, 225, 500, 1324, 3680],SNRdB,0)
     Pb_s_half_7_hard = fec.conv_Pb_bound(1/2,10,[36, 0, 211, 0, 1404, 0, 11633, 0],SNRdB,0)
     Pb_s_half_9_hard = fec.conv_Pb_bound(1/2,12,[33, 0, 281, 0, 2179, 0, 15035, 0],SNRdB,0)
     figure(figsize=(5,5))
     semilogy(SNRdB,Pb_uc)
     semilogy(SNRdB,Pb_s_half_3_hard,'--')
     semilogy(SNRdB,Pb_s_half_5_hard,'--')
     semilogy(SNRdB,Pb_s_half_7_hard,'--')
     semilogy(SNRdB,Pb_s_half_9_hard,'--')
     axis([0,12,1e-7,1e0])
     title(r'Hard Decision Rate 1/2 Coding Theory Bounds')
     xlabel(r'$E_b/N_0$ (dB)')
     ylabel(r'Symbol Error Probability')
     legend(('Uncoded BPSK','R=1/2, K=3, Hard',\
             'R=1/2, K=5, Hard', 'R=1/2, K=7, Hard',\
             'R=1/2, K=9, Hard'),loc='upper right')
     grid();
```

BEP Simulation

Now that we can determine our BEP bounds, we can test the actual encoder/decoder using dummy binary data. The following code creates a rate 1/2 fec_conv object. It then generates dummy binary data and encodes the data using the `conv_encoder` method. This method takes an array of binary values, and an initial state as the input and returns the encoded bits and states. We then adds nois to the encoded data according to the set E_b/N_0 to simulate a noisy channel. The data is then decoded using the `viterbi_decoder` method. This method takes the array of noisy data and a decision metric. If the hard decision metric is selected, then we expect binary input values from around 0 to around 1. The method then returns the decoded binary values. Then the bit errors are counted. Once at least 100 bit errors are counted, the bit error probability is calculated.

```
[7]: N_bits_per_frame = 10000
     EbN0 = 4
     total_bit_errors = 0
     total_bit_count = 0
     cc1 = fec.FECConv(('11101','10011'),25)
     # Encode with shift register starting state of '0000'
     state = '0000'
     while total_bit_errors < 100:
         # Create 100000 random 0/1 bits
         x = randint(0,2,N_bits_per_frame)
         y,state = cc1.conv_encoder(x,state)
         # Add channel noise to bits, include antipodal level shift to [-1,1]
         yn_soft = dc.cpx_awgn(2*y-1,EbN0-3,1) # Channel SNR is 3 dB less for rate 1/2
         yn_hard = ((sign(yn_soft.real)+1)/2).astype(int)
         z = cc1.viterbi_decoder(yn_hard,'hard')
         # Count bit errors
         bit_count, bit_errors = dc.bit_errors(x,z)
         total_bit_errors += bit_errors
         total_bit_count += bit_count
         print('Bits Received = %d, Bit errors = %d, BEP = %1.2e' %\
                 (total_bit_count, total_bit_errors,\
                  total_bit_errors/total_bit_count))
     print('*****************************************************')
     print('Bits Received = %d, Bit errors = %d, BEP = %1.2e' %\
            (total_bit_count, total_bit_errors,\
             total_bit_errors/total_bit_count))

     Bits Received = 9976, Bit errors = 60, BEP = 6.01e-03
     Bits Received = 19952, Bit errors = 137, BEP = 6.87e-03
     *****************************************************
     Bits Received = 19952, Bit errors = 137, BEP = 6.87e-03
```

```
[8]: y[:100].astype(int)
```

```
[8]: array([0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0,
            0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1,
            1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0])
```

The simulated BEP can then be compared to the theoretical bounds that were shown earlier. Some values were simulated for the constraint length 3 and constraint length 5 cases.

```
[9]: SNRdB = arange(0,12,.1)
     Pb_uc = fec.conv_Pb_bound(1/2,7,[4, 12, 20, 72, 225],SNRdB,2)
     Pb_s_half_3_hard = fec.conv_Pb_bound(1/2,5,[1, 4, 12, 32, 80, 192, 448, 1024],SNRdB,0)
     Pb_s_half_5_hard = fec.conv_Pb_bound(1/2,7,[4, 12, 20, 72, 225, 500, 1324, 3680],SNRdB,0)
     Pb_s_half_7_hard = fec.conv_Pb_bound(1/2,10,[36, 0, 211, 0, 1404, 0, 11633, 0],SNRdB,0)
     Pb_s_half_9_hard = fec.conv_Pb_bound(1/2,12,[33, 0, 281, 0, 2179, 0, 15035, 0],SNRdB,0)
     Pb_s_half_5_hard_sim = array([3.36e-2,1.04e-2,1.39e-3,1.56e-04,1.24e-05])
     Pb_s_half_3_hard_sim = array([2.59e-02,1.35e-02,2.71e-03,6.39e-04,9.73e-05,7.71e-06])
     figure(figsize=(5,5))
     semilogy(SNRdB,Pb_uc)
     semilogy(SNRdB,Pb_s_half_3_hard,'y--')
     semilogy(SNRdB,Pb_s_half_5_hard,'g--')
     semilogy(SNRdB,Pb_s_half_7_hard,'--')
     semilogy(SNRdB,Pb_s_half_9_hard,'--')
     semilogy([3,4,5,6,7,8],Pb_s_half_3_hard_sim,'ys')
     semilogy([3,4,5,6,7],Pb_s_half_5_hard_sim,'gs')
     axis([0,12,1e-7,1e0])
     title(r'Hard Decision Rate 1/2 Coding Measurements')
     xlabel(r'$E_b/N_0$ (dB)')
     ylabel(r'Symbol Error Probability')
     legend(('Uncoded BPSK','R=1/2, K=3, Hard',\
             'R=1/2, K=5, Hard', 'R=1/2, K=7, Hard',\
             'R=1/2, K=9, Hard', 'R=1/2, K=3, Simulation',\
            'R=1/2, K=5, Simulation'),loc='lower left')
     grid();
```

We can look at the surviving paths using the `traceback_plot` method.

```
[10]: cc1.traceback_plot()
```

Soft Decision Decoding BEP Simulation

Soft decision decoding can also be done. In order to simulate the soft decision decoder, we can use the same setup as before, but now we specify 'soft' in the `viterbi_decoder` method. We also have to pick a quantization level when we do this. If we want 3-bit quantization we would specify that the quant_level=3. When we use soft decisions we have to scale our noisy received values to values on $[0, 2^n - 1]$. So for a three-bit quantizaiton, we would scale to values on $[0, 7]$. This helps the system to get better distance metrics for all possible paths in the decoder, thus improving the BEP. The following shows how to simulate soft decisions.

```
[11]: N_bits_per_frame = 10000
     EbN0 = 2
     total_bit_errors = 0
     total_bit_count = 0
     cc1 = fec.FECConv(('11101','10011'),25)
     # Encode with shift register starting state of '0000'
     state = '0000'
     while total_bit_errors < 100:
         # Create 100000 random 0/1 bits
         x = randint(0,2,N_bits_per_frame)
         y,state = cc1.conv_encoder(x,state)
```

```python
    # Add channel noise to bits, include antipodal level shift to [-1,1]
    yn = dc.cpx_awgn(2*y-1,EbN0-3,1) # Channel SNR is 3dB less for rate 1/2
    # Scale & level shift to three-bit quantization levels [0,7]
    yn = (yn.real+1)/2*7
    z = cc1.viterbi_decoder(yn.real,'soft',quant_level=3)
    # Count bit errors
    bit_count, bit_errors = dc.bit_errors(x,z)
    total_bit_errors += bit_errors
    total_bit_count += bit_count
    print('Bits Received = %d, Bit errors = %d, BEP = %1.2e' %\
            (total_bit_count, total_bit_errors,\
             total_bit_errors/total_bit_count))
print('*************************************************')
print('Bits Received = %d, Bit errors = %d, BEP = %1.2e' %\
      (total_bit_count, total_bit_errors,\
       total_bit_errors/total_bit_count))
```

```
Bits Received = 9976, Bit errors = 108, BEP = 1.08e-02
*************************************************
Bits Received = 9976, Bit errors = 108, BEP = 1.08e-02
```

```python
[12]: SNRdB = arange(0,12,.1)
      Pb_uc = fec.conv_Pb_bound(1/3,7,[4, 12, 20, 72, 225],SNRdB,2)
      Pb_s_third_3 = fec.conv_Pb_bound(1/3,8,[3, 0, 15],SNRdB,1)
      Pb_s_third_4 = fec.conv_Pb_bound(1/3,10,[6, 0, 6, 0],SNRdB,1)
      Pb_s_third_5 = fec.conv_Pb_bound(1/3,12,[12, 0, 12, 0, 56],SNRdB,1)
      Pb_s_third_6 = fec.conv_Pb_bound(1/3,13,[1, 8, 26, 20, 19, 62],SNRdB,1)
      Pb_s_third_7 = fec.conv_Pb_bound(1/3,14,[1, 0, 20, 0, 53, 0, 184],SNRdB,1)
      Pb_s_third_8 = fec.conv_Pb_bound(1/3,16,[1, 0, 24, 0, 113, 0, 287, 0],SNRdB,1)
      Pb_s_half = fec.conv_Pb_bound(1/2,7,[4, 12, 20, 72, 225],SNRdB,1)
      figure(figsize=(5,5))
      semilogy(SNRdB,Pb_uc)
      semilogy(SNRdB,Pb_s_third_3,'--')
      semilogy(SNRdB,Pb_s_third_4,'--')
      semilogy(SNRdB,Pb_s_third_5,'g')
      semilogy(SNRdB,Pb_s_third_6,'--')
      semilogy(SNRdB,Pb_s_third_7,'--')
      semilogy(SNRdB,Pb_s_third_8,'--')
      #semilogy(SNRdB,Pb_s_half,'--')
      semilogy([0,1,2,3,4,5],[9.08e-02,2.73e-02,6.52e-03,\
                              8.94e-04,8.54e-05,5e-6],'gs')
      axis([0,12,1e-7,1e0])
      title(r'Soft Decision Rate 1/2 Coding Measurements')
      xlabel(r'$E_b/N_0$ (dB)')
      ylabel(r'Symbol Error Probability')
      legend(('Uncoded BPSK','R=1/3, K=3, Soft',\
              'R=1/3, K=4, Soft','R=1/3, K=5, Soft',\
              'R=1/3, K=6, Soft','R=1/3, K=7, Soft',\
              'R=1/3, K=8, Soft','R=1/3, K=5, Sim', \
              'Simulation'),loc='upper right')
      grid();
```

The decoder can also do unquantized soft decisions. This is done by specifying 'unquant' for the metric type. The system will then expect floating point numbers on $[0, 1]$ at the decoder input.

Rate 1/3

Rate 1/3 convolution encoding/decoding can be done very similarly to the rate 1/2 code. The difference when instantiating, is that the rate 1/3 uses 3 generator polynmials instead of 2. The following table shows ideal generator polynomials at different constraint lengths for rate 1/3 convolutional codes.

Table 2: Weight spectra :math:`c_k` for bounding the coded rate 1/3 BEP.

CL	Polynomials	d_{free}	d_f	$d_f + 1$	$d_f + 2$	$d_f + 3$	$d_f + 4$	$d_f + 5$	$d_f + 6$	$d_f + 7$
3	(7,7,5) = ('111','111','101')	8	3	0	15	0	58	0	201	0
4	(15,13,11) = ('1111','1101','1011')	10	6	0	6	0	58	0	118	0
5	(31,27,21) = ('11111','11011','10101')	12	12	0	12	0	56	0	320	0
6	(61,43,39) = ('111101','101011','100111')	13	1	8	26	20	19	62	86	204
7	(121,101,91) = ('1111001','1100101','1011011')	14	1	0	20	0	53	0	184	0
8	(247,217,149) = ('11110111','11011001','10010101')	16	1	0	24	0	113	0	287	0

```
[13]: cc2 = fec.FECConv(('111','111','101'),10)
      cc2.trellis_plot()
```

```
/home/docs/.pyenv/versions/3.7.9/lib/python3.7/site-packages/numpy/core/_asarray.py:136:
→VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is
→a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is
→deprecated. If you meant to do this, you must specify 'dtype=object' when creating the
→ndarray
  return array(a, dtype, copy=False, order=order, subok=True)
```

Rate 1/3 Hard Decision Decoding

Weight Structure Bounds BEP

Compare with Ziemer pg 668.

```
[14]: SNRdB = arange(0,12,.1)
      Pb_uc = fec.conv_Pb_bound(1/3,7,[4, 12, 20, 72, 225],SNRdB,2)
      Pb_s_third_3_hard = fec.conv_Pb_bound(1/3,8,[3, 0, 15, 0, 58, 0, 201, 0],SNRdB,0)
      Pb_s_third_4_hard = fec.conv_Pb_bound(1/3,10,[6, 0, 6, 0, 58, 0, 118, 0],SNRdB,0)
      Pb_s_third_5_hard = fec.conv_Pb_bound(1/3,12,[12, 0, 12, 0, 56, 0, 320, 0],SNRdB,0)
      Pb_s_third_6_hard = fec.conv_Pb_bound(1/3,13,[1, 8, 26, 20, 19, 62, 86, 204],SNRdB,0)
      Pb_s_third_7_hard = fec.conv_Pb_bound(1/3,14,[1, 0, 20, 0, 53, 0, 184],SNRdB,0)
      Pb_s_third_8_hard = fec.conv_Pb_bound(1/3,16,[1, 0, 24, 0, 113, 0, 287, 0],SNRdB,0)
      figure(figsize=(5,5))
      semilogy(SNRdB,Pb_uc)
```

```
semilogy(SNRdB,Pb_s_third_3_hard,'--')
#semilogy(SNRdB,Pb_s_third_4_hard,'--')
semilogy(SNRdB,Pb_s_third_5_hard,'--')
#semilogy(SNRdB,Pb_s_third_6_hard,'--')
semilogy(SNRdB,Pb_s_third_7_hard,'--')
#semilogy(SNRdB,Pb_s_third_8_hard,'--')
axis([0,12,1e-7,1e0])
title(r'Hard Decision Rate 1/3 Coding Theory Bounds')
xlabel(r'$E_b/N_0$ (dB)')
ylabel(r'Symbol Error Probability')
legend(('Uncoded BPSK','R=1/3, K=3, Hard',\
        #'R=1/3, K=4, Hard', 'R=1/3, K=5, Hard',\
        #'R=1/3, K=6, Hard', 'R=1/3, K=7, Hard',\
        #'R=1/3, K=7, Hard'),loc='upper right')
        'R=1/3, K=5, Hard', 'R=1/3, K=7, Hard'),\
        loc='upper right')
grid();
```

BEP Simulation

```
[15]: N_bits_per_frame = 10000
      EbN0 = 3
      total_bit_errors = 0
      total_bit_count = 0
      cc1 = fec.FECConv(('11111','11011','10101'),25)
      # Encode with shift register starting state of '0000'
      state = '0000'
      while total_bit_errors < 100:
          # Create 100000 random 0/1 bits
          x = randint(0,2,N_bits_per_frame)
          y,state = cc1.conv_encoder(x,state)
          # Add channel noise to bits, include antipodal level shift to [-1,1]
          yn_soft = dc.cpx_awgn(2*y-1,EbN0-10*log10(3),1) # Channel SNR is 10*log10(3) dB less
          yn_hard = ((sign(yn_soft.real)+1)/2).astype(int)
          z = cc1.viterbi_decoder(yn_hard.real,'hard')
          # Count bit errors
          bit_count, bit_errors = dc.bit_errors(x,z)
          total_bit_errors += bit_errors
          total_bit_count += bit_count
          print('Bits Received = %d, Bit errors = %d, BEP = %1.2e' %\
                (total_bit_count, total_bit_errors,\
                 total_bit_errors/total_bit_count))
      print('*****************************************************')
      print('Bits Received = %d, Bit errors = %d, BEP = %1.2e' %\
            (total_bit_count, total_bit_errors,\
             total_bit_errors/total_bit_count))
```

```
Bits Received = 9976, Bit errors = 181, BEP = 1.81e-02
*****************************************************
Bits Received = 9976, Bit errors = 181, BEP = 1.81e-02
```

```
[16]: SNRdB = arange(0,12,.1)
      Pb_uc = fec.conv_Pb_bound(1/3,7,[4, 12, 20, 72, 225],SNRdB,2)
      Pb_s_third_3_hard = fec.conv_Pb_bound(1/3,8,[3, 0, 15, 0, 58, 0, 201, 0],SNRdB,0)
      Pb_s_third_5_hard = fec.conv_Pb_bound(1/3,12,[12, 0, 12, 0, 56, 0, 320, 0],SNRdB,0)
      Pb_s_third_7_hard = fec.conv_Pb_bound(1/3,14,[1, 0, 20, 0, 53, 0, 184],SNRdB,0)
      Pb_s_third_5_hard_sim = array([8.94e-04,1.11e-04,8.73e-06])
      figure(figsize=(5,5))
      semilogy(SNRdB,Pb_uc)
      semilogy(SNRdB,Pb_s_third_3_hard,'r--')
      semilogy(SNRdB,Pb_s_third_5_hard,'g--')
      semilogy(SNRdB,Pb_s_third_7_hard,'k--')
      semilogy(array([5,6,7]),Pb_s_third_5_hard_sim,'sg')
      axis([0,12,1e-7,1e0])
      title(r'Hard Decision Rate 1/3 Coding Measurements')
      xlabel(r'$E_b/N_0$ (dB)')
      ylabel(r'Symbol Error Probability')
      legend(('Uncoded BPSK','R=1/3, K=3, Hard',\
              'R=1/3, K=5, Hard', 'R=1/3, K=7, Hard',\
              ),loc='upper right')
      grid();
```

```
[17]: cc1.traceback_plot()
```

Soft Decision Decoding BEP Simulation

Here we use 3-bit quantization soft decoding.

```
[18]: N_bits_per_frame = 10000
      EbN0 = 2
      total_bit_errors = 0
      total_bit_count = 0
      cc1 = fec.FECConv(('11111','11011','10101'),25)
      # Encode with shift register starting state of '0000'
      state = '0000'
      while total_bit_errors < 100:
          # Create 100000 random 0/1 bits
          x = randint(0,2,N_bits_per_frame)
          y,state = cc1.conv_encoder(x,state)
          # Add channel noise to bits, include antipodal level shift to [-1,1]
          yn = dc.cpx_awgn(2*y-1,EbN0-10*log10(3),1) # Channel SNR is 10*log10(3) dB less
          # Translate to [0,7]
          yn = (yn.real+1)/2*7
          z = cc1.viterbi_decoder(yn,'soft',quant_level=3)
          # Count bit errors
          bit_count, bit_errors = dc.bit_errors(x,z)
          total_bit_errors += bit_errors
          total_bit_count += bit_count
          print('Bits Received = %d, Bit errors = %d, BEP = %1.2e' %\
                (total_bit_count, total_bit_errors,\
                 total_bit_errors/total_bit_count))
```

```python
print('******************************************************')
print('Bits Received = %d, Bit errors = %d, BEP = %1.2e' %\
        (total_bit_count, total_bit_errors,\
         total_bit_errors/total_bit_count))
```

```
Bits Received = 9976, Bit errors = 56, BEP = 5.61e-03
Bits Received = 19952, Bit errors = 87, BEP = 4.36e-03
Bits Received = 29928, Bit errors = 170, BEP = 5.68e-03
******************************************************
Bits Received = 29928, Bit errors = 170, BEP = 5.68e-03
```

```python
[19]: SNRdB = arange(0,12,.1)
      Pb_uc = fec.conv_Pb_bound(1/3,7,[4, 12, 20, 72, 225],SNRdB,2)
      Pb_s_third_3 = fec.conv_Pb_bound(1/3,8,[3, 0, 15, 0, 58, 0, 201, 0],SNRdB,1)
      #Pb_s_third_4 = fec.conv_Pb_bound(1/3,10,[6, 0, 6, 0, 58, 0, 118, 0],SNRdB,1)
      Pb_s_third_5 = fec.conv_Pb_bound(1/3,12,[12, 0, 12, 0, 56, 0, 320, 0],SNRdB,1)
      #Pb_s_third_6 = fec.conv_Pb_bound(1/3,13,[1, 8, 26, 20, 19, 62, 86, 204],SNRdB,1)
      Pb_s_third_7 = fec.conv_Pb_bound(1/3,14,[1, 0, 20, 0, 53, 0, 184, 0],SNRdB,1)
      #Pb_s_third_8 = fec.conv_Pb_bound(1/3,16,[1, 0, 24, 0, 113, 0, 287, 0],SNRdB,1)
      figure(figsize=(5,5))
      semilogy(SNRdB,Pb_uc)
      semilogy(SNRdB,Pb_s_third_3,'--')
      #semilogy(SNRdB,Pb_s_third_4,'--')
      semilogy(SNRdB,Pb_s_third_5,'g')
      #semilogy(SNRdB,Pb_s_third_6,'--')
      semilogy(SNRdB,Pb_s_third_7,'r--')
      #semilogy(SNRdB,Pb_s_third_8,'--')
      #semilogy(SNRdB,Pb_s_half,'--')
      semilogy([0,1,2,3,4,5],[9.08e-02,2.73e-02,6.52e-03,\
                              8.94e-04,8.54e-05,5e-6],'gs')
      axis([0,12,1e-7,1e0])
      title(r'Soft Decision Rate 1/3 Coding Measurements')
      xlabel(r'$E_b/N_0$ (dB)')
      ylabel(r'Symbol Error Probability')
      legend(('Uncoded BPSK','R=1/3, K=3, Soft',\
              #'R=1/3, K=4, Soft','R=1/3, K=5, Soft',\
              'R=1/3, K=5, Soft','R=1/3, K=7, Soft',\
              #'R=1/3, K=8, Soft','R=1/2, K=5, Soft', \
              'R-1/3, K=5, Simulation'),loc='upper right')
      grid();
```

```
[1]: %pylab inline
     #%matplotlib qt
     import sk_dsp_comm.sigsys as ss
     import scipy.signal as signal
     from IPython.display import Audio, display
     from IPython.display import Image, SVG
```

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

```
[2]: 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
```

```
[3]: import scipy.special as special
     import sk_dsp_comm.digitalcom as dc
     import sk_dsp_comm.fec_block as block
```

Block Codes

Block codes take serial source symbols and group them into k-symbol blocks. They then take n-k check symbols to make code words of length n > k. The code is denoted (n,k). The following shows a general block diagram of block encoder.

Input → Upper register → $+$ → Output
Switch B
AND gate
Complement
C C
Switch A → $+$ → S_1 → S_2 → S_3
$+$

Received word 1101001 (no errors)

	Shift	Input	Lower register content	Output
Switch A closed Switch B open	1	1	100	
	2	1	110	
	3	0	111	
	4	1	111	
	5	0	011	
	6	0	001	
	7	1	000	
Switch A open Switch B closed	8		000	1
	9		000	1
	10		000	0
	11		000	1
	12		000	0
	13		000	0
	14		000	1

Received word 1101011 (one errors)

Shift	Input	Lower register content	Output
1	1	100	
2	1	110	
3	0	111	
4	1	111	
5	0	011	
6	1	101	
7	1	010	
8		101	1
9		110	1
10		111	0
11		011	1
12		001	0
13		100	0
14		010	1

AND gate output inverts upper register output

The block encoder takes k source bits and encodes it into a length n codeword. A block decoder then works in reverse. The length n channel symbol codewords are decoded into the original length k source bits.

Single Error Correction Block Codes

Several block codes are able to correct only one error per block. Two common single error correction codes are cyclic codes and hamming codes. In `scikit-dsp-comm` there is a module called `fec_block.py`. This module contains two classes so far: `fec_cyclic` for cyclic codes and `fec_hamming` for hamming codes. Each class has methods for encoding, decoding, and plotting theoretical bit error probability bounds.

Cyclic Codes

A (n,k) cyclic code can easily be generated with an n-k stage shift register with appropriate feedback according to Ziemer and Tranter pgs 646 and 647. The following shows a block diagram for a cyclic encoder.

Input

Upper register

Output

Switch B

AND gate

Complement

C C

Switch A

S_1 S_2 S_3

Received word 1101001 (no errors)

Shift	Input	Lower register content	Output
1	1	100	
2	1	110	
3	0	111	
4	1	111	
5	0	011	
6	0	001	
7	1	000	
8		000	1
9		000	1
10		000	0
11		000	1
12		000	0
13		000	0
14		000	1

Switch A closed Switch B open

Switch A open Switch B closed

Received word 1101011 (one errors)

Shift	Input	Lower register content	Output
1	1	100	
2	1	110	
3	0	111	
4	1	111	
5	0	011	
6	1	101	
7	1	010	
8		101	1
9		110	1
10		111	0
11		011	1
12		001	0
13		100	0
14		010	1

AND gate output inverts upper register output

This block diagram can be expanded to larger codes as well. A generator polynomial can be used to determine the position of the binary adders. The previous example uses a generator polynomial of '1011'. This means that there is a binary adder after the input, after second shift register, and after the third shift register.

The source symbol length and the channel symbol length can be determined from the number of shift registers j. The length of the generator polynomial is always $1 + j$. In this case we have 3 shift registers, so $j = 3$. We have $k = 4$ source bits and $n = 7$ channel bits. For other shift register lengths, we can use the following equations. $n = j^2 - 1$ and $k = n - j$. The following table (from Ziemer and Peterson pg 429) shows the source symbol length, channel symbol

length, and the code rate for various shift register lengths for single error correction codes.

j	k	n	R=k/n
3	4	7	0.57
4	11	15	0.73
5	26	31	0.84
6	57	63	0.90
7	120	127	0.94
8	247	255	0.97
9	502	511	0.98
10	1013	1023	0.99

The following block diagram shows a block decoder (from Ziemer and Tranter page 647). The block decoder takes in a codeword of channel symbol length n and decodes it to the original source bits of length k.

Received word 1101001 (no errors)

	Shift	Input	Lower register content	Output
Switch A closed Switch B open	1	1	100	
	2	1	110	
	3	0	111	
	4	1	111	
	5	0	011	
	6	0	001	
	7	1	000	
Switch A open Switch B closed	8		000	1
	9		000	1
	10		000	0
	11		000	1
	12		000	0
	13		000	0
	14		000	1

Received word 1101011 (one errors)

Shift	Input	Lower register content	Output
1	1	100	
2	1	110	
3	0	111	
4	1	111	
5	0	011	
6	1	101	
7	1	010	
8		101	1
9		110	1
10		111	0
11		011	1
12		001	0
13		100	0
14		010	1

AND gate output inverts upper register output

The `fec_cyclic` class can be used to generate a cyclic code object. The cyclic code object can be initialized by a generator polynomial. The length of the generator determines the source symbol length, the channel symbol length, and the rate. The following shows the generator polynomial '1011' considered in the two example block diagrams.

```
[4]: cc1 = block.FECCyclic('1011')
```

After the cyclic code object `cc1` is created, the `cc1.cyclic_encoder` method can be used to encode source data bits. In the following example, we generate 16 distinct source symbols to get 16 distinct channel symbol codewords using

the `cyclic_encoder` method. The `cyclic_encoder` method takes an array of source bits as a paramter. The array of source bits must be a length of a multiple of k. Otherwise, the method will throw an error.

```
[5]: # Generate 16 distinct codewords
     codewords = zeros((16,7),dtype=int)
     x = zeros((16,4))
     for i in range(0,16):
         xbin = block.binary(i,4)
         xbin = array(list(xbin)).astype(int)
         x[i,:] = xbin
     x = reshape(x,size(x)).astype(int)
     codewords = cc1.cyclic_encoder(x)
     print(reshape(codewords,(16,7)))
```

```
[[0 0 0 0 0 0 0]
 [0 0 0 1 0 1 1]
 [0 0 1 0 1 1 0]
 [0 0 1 1 1 0 1]
 [0 1 0 0 1 1 1]
 [0 1 0 1 1 0 0]
 [0 1 1 0 0 0 1]
 [0 1 1 1 0 1 0]
 [1 0 0 0 1 0 1]
 [1 0 0 1 1 1 0]
 [1 0 1 0 0 1 1]
 [1 0 1 1 0 0 0]
 [1 1 0 0 0 1 0]
 [1 1 0 1 0 0 1]
 [1 1 1 0 1 0 0]
 [1 1 1 1 1 1 1]]
```

Now, a bit error is introduced into each of the codewords. Then, the codwords with the error are decoded using the `cyclic_decoder` method. The `cyclic_decoder` method takes an array of codewords of length n as a parameter and returns an array of source bits. Even with 1 error introduced into each codeword, All of the original source bits are still decoded properly.

```
[6]: # introduce 1 bit error into each code word and decode
     codewords = reshape(codewords,(16,7))
     for i in range(16):
         error_pos = i % 6
         codewords[i,error_pos] = (codewords[i,error_pos] +1) % 2
     codewords = reshape(codewords,size(codewords))
     decoded_blocks = cc1.cyclic_decoder(codewords)
     print(reshape(decoded_blocks,(16,4)))
```

```
[[0 0 0 0]
 [0 0 0 1]
 [0 0 1 0]
 [0 0 1 1]
 [0 1 0 0]
 [0 1 0 1]
 [0 1 1 0]
 [0 1 1 1]
 [1 0 0 0]
 [1 0 0 1]
```

```
 [1 0 1 0]
 [1 0 1 1]
 [1 1 0 0]
 [1 1 0 1]
 [1 1 1 0]
 [1 1 1 1]]
```

The following example generates many random source symbols. It then encodes the symbols using the cyclic encoder. It then simulates a channel by adding noise. It then implements hard decisions on each of the incoming bits and puts the received noisy bits into the cyclic decoder. Source bits are then returned and errors are counted until 100 bit errors are received. Once 100 bit errors are received, the bit error probability is calculated. This code can be run at a variety of SNRs and with various code rates.

```python
[7]: cc1 = block.FECCyclic('101001')
     N_blocks_per_frame = 2000
     N_bits_per_frame = N_blocks_per_frame*cc1.k
     EbN0 = 6
     total_bit_errors = 0
     total_bit_count = 0

     while total_bit_errors < 100:
         # Create random 0/1 bits
         x = randint(0,2,N_bits_per_frame)
         y = cc1.cyclic_encoder(x)
         # Add channel noise to bits and scale to +/- 1
         yn = dc.cpx_awgn(2*y-1,EbN0-10*log10(cc1.n/cc1.k),1) # Channel SNR is dB less
         # Scale back to 0 and 1
         yn = ((sign(yn.real)+1)/2).astype(int)
         z = cc1.cyclic_decoder(yn)
         # Count bit errors
         bit_count, bit_errors = dc.bit_errors(x,z)
         total_bit_errors += bit_errors
         total_bit_count += bit_count
         print('Bits Received = %d, Bit errors = %d, BEP = %1.2e' %\
                 (total_bit_count, total_bit_errors,\
                  total_bit_errors/total_bit_count))
     print('*************************************************')
     print('Bits Received = %d, Bit errors = %d, BEP = %1.2e' %\
             (total_bit_count, total_bit_errors,\
              total_bit_errors/total_bit_count))
```

```
Bits Received = 52000, Bit errors = 27, BEP = 5.19e-04
Bits Received = 104000, Bit errors = 83, BEP = 7.98e-04
Bits Received = 156000, Bit errors = 141, BEP = 9.04e-04
*************************************************
Bits Received = 156000, Bit errors = 141, BEP = 9.04e-04
```

There is a function in the `fec_block` module called `block_single_error_Pb_bound` that can be used to generate the theoretical bit error probability bounds for single error correction block codes. Measured bit error probabilities from the previous example were recorded to compare to the bounds.

```python
[8]: SNRdB = arange(0,12,.1)
     #SNRdB = arange(9.4,9.6,0.1)
```

```
Pb_uc = block.block_single_error_Pb_bound(3,SNRdB,False)
Pb_c_3 = block.block_single_error_Pb_bound(3,SNRdB)
Pb_c_4 = block.block_single_error_Pb_bound(4,SNRdB)
Pb_c_5 = block.block_single_error_Pb_bound(5,SNRdB)
figure(figsize=(5,5))
semilogy(SNRdB,Pb_uc,'k-')
semilogy(SNRdB,Pb_c_3,'c--')
semilogy(SNRdB,Pb_c_4,'m--')
semilogy(SNRdB,Pb_c_5,'g--')
semilogy([4,5,6,7,8,9],[1.44e-2,5.45e-3,2.37e-3,6.63e-4,1.33e-4,1.31e-5],'cs')
semilogy([5,6,7,8],[4.86e-3,1.16e-3,2.32e-4,2.73e-5],'ms')
semilogy([5,6,7,8],[4.31e-3,9.42e-4,1.38e-4,1.15e-5],'gs')
axis([0,12,1e-10,1e0])
title('Cyclic code BEP')
xlabel(r'$E_b/N_0$ (dB)')
ylabel(r'Bit Error Probability')
legend(('Uncoded BPSK','(7,4), hard',\
        '(15,11), hard', '(31,26), hard',\
        '(7,4) sim', '(15,11) sim', \
        '(31,26) sim'),loc='lower left')
grid();
```

These plots show that the simulated bit error probability is very close to the theoretical bit error probabilites.

Hamming Code

Hamming codes are another form of single error correction block codes. Hamming codes use parity-checks in order to generate and decode block codes. The code rates of Hamming codes are generated the same way as cyclic codes. In this case a parity-check length of length j is chosen, and n and k are calculated by $n = 2^j - 1$ and $k = n - j$. Hamming codes are generated first by defining a parity-check matrix H. The parity-check matrix is a j x n matrix containing binary numbers from 1 to n as the columns. For a $j = 3$ ($k = 4$, $n = 7$) Hamming code. The parity-check matrix starts out as the following:

$$\mathbf{H} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & & \\ 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & & \\ 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & & \end{bmatrix} \tag{2.40}$$

The parity-chekc matrix can be reordered to provice a systematic code by interchanging the columns to create an identity matrix on the right side of the matrix. In this case, this is done by interchangeing columsn 1 and 7, columns 2 and 6,

and columsn 4 and 5. The resulting parity-check matrix is the following.

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & & \\ 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & & \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & & \end{bmatrix} \tag{2.41}$$

Next, a generator matrix G is created by restructuring the parity-check matrix. The G matrix is gathered from the H matrix through the following relationship.

$$\mathbf{G} = \begin{bmatrix} I_k & ... & H_p \end{bmatrix} \tag{2.42}$$

where H_p is defined as the transpose of the first k columns of H. For this example we arrive at the following G matrix. G always ends up being a k x n matrix.

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & & \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & & \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & & \\ 0 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & & \end{bmatrix} \tag{2.43}$$

Codewords can be generated by multiplying a source symbol matrix by the generator matrix.

$$codeword = xG \tag{2.44}$$

Where the codeword is a column vector of length n and x is a row vector of length n. This is the basic operation of the encoder. The decoder is slightly more complicated. The decoder starts by taking the parity-check matrix H and multiplying it by the codeword column vector. This gives the "syndrome" of the block. The syndrome tells us whether or not there is an error in the codeword. If no errors are present, the syndrome will be 0. If there is an error in the codeword, the syndrome will tell us which bit has the error.

$$S = H \cdot codeword \tag{2.45}$$

If the syndrome is nonzero, then it can be used to correct the error bit in the codeword. After that, the original source blocks can be decoded from the codewords by the following equation.

$$source = R \cdot codeword \tag{2.46}$$

Where R is a k x n matrix where R is made up of a k x k identity matrix and a k x n-k matrix of zeros. Again, the Hamming code is only capable of correcting one error per block, so if more than one error is present in the block, then the syndrome cannot be used to correct the error.

The hamming code class can be found in the `fec_block` module as `fec_hamming`. Hamming codes are sometimes generated using generator polynomials just like with cyclic codes. This is not completely necessary, however, if the previously described process is used. This process simply relies on choosing a number of parity bits and then systematic single-error correction hamming codes are automatically generated. The following will go through an example of a $j = 3$ ($k = 4$, $n = 7$) hamming code.

Hamming Block Code Class Definition:

```
[9]: hh1 = block.FECHamming(3)
```

k and n are calculated form the number of parity checks j and can be accessed by `hh1.k` and `hh1.n`. The j x n parity-check matrix H and the k x n generator matrix G can be accessed by `hh1.H` and `hh1.G`. These are exactly as described previously.

```
[10]: print('k = ' + str(hh1.k))
      print('n = ' + str(hh1.n))
      print('H = \n' + str(hh1.H))
      print('G = \n' + str(hh1.G))

k = 4
n = 7
H =
[[1 1 0 1 1 0 0]
 [1 1 1 0 0 1 0]
 [1 0 1 1 0 0 1]]
G =
[[1 0 0 0 1 1 1]
 [0 1 0 0 1 1 0]
 [0 0 1 0 0 1 1]
 [0 0 0 1 1 0 1]]
```

The `fec_hamming` class has an encoder method called `hamm_encoder`. This method works the same way as the cyclic encoder. It takes an array of source bits with a length that is a multiple of k and returns an array of codewords. This class has another method called `hamm_decoder` which can decode an array of codewords. The array of codewords must have a length that is a multiple of n. The following example generates random source bits, encodes them using a hamming encoder, simulates transmitting them over a channel, uses hard decisions after the receiver to get a received array of codewords, and decodes the codewords using the hamming decoder. It runs until it counds 100 bit errors and then calculates the bit error probability. This can be used to simulate hamming codes with different rates (different numbers of parity checks) at different SNRs.

```
[11]: hh1 = block.FECHamming(5)
      N_blocks_per_frame = 20000
      N_bits_per_frame = N_blocks_per_frame*hh1.k
      EbN0 = 8
      total_bit_errors = 0
      total_bit_count = 0


      while total_bit_errors < 100:
          # Create random 0/1 bits
          x = randint(0,2,N_bits_per_frame)
          y = hh1.hamm_encoder(x)
          # Add channel noise to bits and scale to +/- 1
          yn = dc.cpx_awgn(2*y-1,EbN0-10*log10(hh1.n/hh1.k),1) # Channel SNR is dB less
          # Scale back to 0 and 1
          yn = ((sign(yn.real)+1)/2).astype(int)
          z = hh1.hamm_decoder(yn)
          # Count bit errors
          bit_count, bit_errors = dc.bit_errors(x,z)
          total_bit_errors += bit_errors
          total_bit_count += bit_count
          print('Bits Received = %d, Bit errors = %d, BEP = %1.2e' %\
```

```
            (total_bit_count, total_bit_errors,\
             total_bit_errors/total_bit_count))
print('**************************************************')
print('Bits Received = %d, Bit errors = %d, BEP = %1.2e' %\
        (total_bit_count, total_bit_errors,\
         total_bit_errors/total_bit_count))
```

```
Bits Received = 520000, Bit errors = 8, BEP = 1.54e-05
Bits Received = 1040000, Bit errors = 20, BEP = 1.92e-05
Bits Received = 1560000, Bit errors = 26, BEP = 1.67e-05
Bits Received = 2080000, Bit errors = 26, BEP = 1.25e-05
Bits Received = 2600000, Bit errors = 34, BEP = 1.31e-05
Bits Received = 3120000, Bit errors = 42, BEP = 1.35e-05
Bits Received = 3640000, Bit errors = 48, BEP = 1.32e-05
Bits Received = 4160000, Bit errors = 50, BEP = 1.20e-05
Bits Received = 4680000, Bit errors = 56, BEP = 1.20e-05
Bits Received = 5200000, Bit errors = 63, BEP = 1.21e-05
Bits Received = 5720000, Bit errors = 66, BEP = 1.15e-05
Bits Received = 6240000, Bit errors = 76, BEP = 1.22e-05
Bits Received = 6760000, Bit errors = 77, BEP = 1.14e-05
Bits Received = 7280000, Bit errors = 86, BEP = 1.18e-05
Bits Received = 7800000, Bit errors = 96, BEP = 1.23e-05
Bits Received = 8320000, Bit errors = 107, BEP = 1.29e-05
**************************************************
Bits Received = 8320000, Bit errors = 107, BEP = 1.29e-05
```

The `fec_block.block_single_error_Pb_bound` function can also be used to generate the bit error probability bounds for hamming codes. The following example generates theoretical bit error probability bounds for hamming codes and compares it with simulated bit error probabilities from the previous examples.

```
[12]: SNRdB = arange(0,12,.1)
      Pb_uc = block.block_single_error_Pb_bound(3,SNRdB,False)
      Pb_c_3 = block.block_single_error_Pb_bound(3,SNRdB)
      Pb_c_4 = block.block_single_error_Pb_bound(4,SNRdB)
      Pb_c_5 = block.block_single_error_Pb_bound(5,SNRdB)
      figure(figsize=(5,5))
      semilogy(SNRdB,Pb_uc,'k-')
      semilogy(SNRdB,Pb_c_3,'c--')
      semilogy(SNRdB,Pb_c_4,'m--')
      semilogy(SNRdB,Pb_c_5,'g--')
      semilogy([5,6,7,8,9,10],[6.64e-3,2.32e-3,5.25e-4,1.16e-4,1.46e-5,1.19e-6],'cs')
      semilogy([5,6,7,8,9],[4.68e-3,1.19e-3,2.48e-4,3.6e-5,1.76e-6],'ms')
      semilogy([5,6,7,8,9],[4.42e-3,1.11e-3,1.41e-4,1.43e-5,6.73e-7],'gs')
      axis([0,12,1e-10,1e0])
      title('Hamming code BEP')
      xlabel(r'$E_b/N_0$ (dB)')
      ylabel(r'Bit Error Probability')
      legend(('Uncoded BPSK','(7,4), hard',\
              '(15,11), hard', '(31,26), hard',\
              '(7,4) sim', '(15,11) sim', \
              '(31,26) sim'),loc='lower left')
      grid();
```

Multiple Error Correction Block Codes

Other block codes are capable of correcting multiple errors in blocks. Golay Codes, Bose_Chaudhuri-Hocquenghem (BCH) Codes, and Reed-Solomon Codes are all capable of correcting multiple errors. These codes have not been developed yet, but they will be the next codes to be added to the `fec_block` module.

Golay Code

Golay codes are capable of correcting three errors in a block of 23 symbols. Golay codes are one of the few known "perfect" codes where all error patterns with hamming weight t or less and no error patters with weight greater than t are correctable using a minimum-distance maximum-likelihood decoder. Golay codes are discussed in detail in Ziemer and Peterson pgs 448-450.

Bose-Chaudhuri-Hocquenghem (BCH) Codes

BCH codes are very important because they exist for a wide range of rates, can achieve significant coding gain, and decoders can be implemented even at high speeds. BCH codes are described in detail in Ziemer and Peterson pgs 436-444.

Reed-Solomon Codes

RS codes are nonbinary BCH codes that use input and output alphabets having 2^m symbols, $\{0, 1, 2, ..., 2^m - 1\}$. Block length is $n = 2^m - 1$ and can be extended to $n = 2^m$ or $n = 2^m + 1$. Reed-Solomon codes are useful in burst communications Reed-Solomon Codes are discussed in detail in Ziemer and Peterson pgs 444-447.

coeff2header

Digital Filter Coefficient Conversion to C Header Files

The views and conclusions contained in the software and documentation are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the FreeBSD Project.

sk_dsp_comm.coeff2header.**ca_code_header**(*fname_out*, *Nca*)
> Write 1023 bit CA (Gold) Code Header Files

> Mark Wickert February 2015

sk_dsp_comm.coeff2header.**fir_fix_header**(*fname_out*, *h*)
> Write FIR Fixed-Point Filter Header Files

> Mark Wickert February 2015

sk_dsp_comm.coeff2header.**fir_header**(*fname_out*, *h*)
> Write FIR Filter Header Files

> Mark Wickert February 2015

sk_dsp_comm.coeff2header.**freqz_resp_list**(*b*, *a=array([1])*, *mode='dB'*, *fs=1.0*, *n_pts=1024*, *fsize=(6, 4)*)
> A method for displaying digital filter frequency response magnitude, phase, and group delay. A plot is produced using matplotlib

> freq_resp(self,mode = 'dB',Npts = 1024)

> A method for displaying the filter frequency response magnitude, phase, and group delay. A plot is produced using matplotlib

> freqz_resp(b,a=[1],mode = 'dB',Npts = 1024,fsize=(6,4))

> > **Parameters**

> > > **b** [ndarray of numerator coefficients]

> > > **a** [ndarray of denominator coefficents]

> > > **mode** [display mode: 'dB' magnitude, 'phase' in radians, or] 'groupdelay_s' in samples and 'groupdelay_t' in sec, all versus frequency in Hz

> > > **n_pts** [number of points to plot; default is 1024]

> > > **fsize** [figure size; defult is (6,4) inches]

> > > **Mark Wickert, January 2015**

sk_dsp_comm.coeff2header.**iir_sos_header**(*fname_out*, *SOS_mat*)
> Write IIR SOS Header Files File format is compatible with CMSIS-DSP IIR Directform II Filter Functions

> Mark Wickert March 2015-October 2016

digitalcom

Digital Communications Function Module

Copyright (c) March 2017, Mark Wickert All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

sk_dsp_comm.digitalcom.**awgn_channel**(*x_bits*, *eb_n0_dB*)

> **Parameters**
>
>> **x_bits** [serial bit stream of 0/1 values.]
>>
>> **eb_n0_dB** [Energy per bit to noise power density ratio in dB of the serial bit stream sent through the AWGN channel. Frequently we equate EBN0 to SNR in link budget calculations.]
>
> **Returns**
>
>> **y_bits** [Received serial bit stream following hard decisions. This bit will have bit errors. To check the estimated bit error probability use BPSK_BEP() or simply:]

```
>>> Pe_est = sum(xor(x_bits,y_bits))/length(x_bits);
    ..
```

> **Mark Wickert, March 2015**

sk_dsp_comm.digitalcom.**bin2gray**(*d_word*, *b_width*)

> Convert integer bit words to gray encoded binary words via Gray coding starting from the MSB to the LSB
>
> Mark Wickert November 2018

sk_dsp_comm.digitalcom.**bit_errors**(*tx_data*, *rx_data*, *n_corr=1024*, *n_transient=0*)

> Count bit errors between a transmitted and received BPSK signal. Time delay between streams is detected as well as ambiguity resolution due to carrier phase lock offsets of $k * \pi$, k=0,1.

sk_dsp_comm.digitalcom.**bpsk_bep**(*tx_data*, *rx_data*, *n_corr=1024*, *n_transient=0*)

> Count bit errors between a transmitted and received BPSK signal. Time delay between streams is detected as well as ambiguity resolution due to carrier phase lock offsets of $k * \pi$, k=0,1. The ndarray tx_data is Tx +/-1 symbols as real numbers I. The ndarray rx_data is Rx +/-1 symbols as real numbers I. Note: Ncorr needs to be even

sk_dsp_comm.digitalcom.**bpsk_tx**(*n_bits*, *ns*, *ach_fc=2.0*, *ach_lvl_dB=- 100*, *pulse='rect'*, *alpha=0.25*, *m=6*)

> Generates biphase shift keyed (BPSK) transmitter with adjacent channel interference.
>
> Generates three BPSK signals with rectangular or square root raised cosine (SRC) pulse shaping of duration N_bits and Ns samples per bit. The desired signal is centered on f = 0, which the adjacent channel signals to the left and right are also generated at dB level relative to the desired signal. Used in the digital communications Case Study supplement.
>
> > **Parameters**
> >
> > > **n_bits** [the number of bits to simulate]
> > >
> > > **ns** [the number of samples per bit]

ach_fc [the frequency offset of the adjacent channel signals (default 2.0)]

ach_lvl_dB [the level of the adjacent channel signals in dB (default -100)]

pulse [the pulse shape 'rect' or 'src']

alpha [square root raised cosine pulse shape factor (default = 0.25)]

m [square root raised cosine pulse truncation factor (default = 6)]

Returns

x [ndarray of the composite signal x0 + ach_lvl*(x1p + x1m)]

b [the transmit pulse shape]

data0 [the data bits used to form the desired signal; used for error checking]

Examples

```
>>> x,b,data0 = bpsk_tx(1000,10,pulse='src')
```

sk_dsp_comm.digitalcom.**chan_est_equalize**(*z*, *npbp*, *alpha*, *ht=None*)

This is a helper function for `OFDM_rx()` to unpack pilot blocks from from the entire set of received OFDM symbols (the Nf of N filled carriers only); then estimate the channel array H recursively, and finally apply H_hat to Y, i.e., X_hat = Y/H_hat carrier-by-carrier. Note if Np = -1, then H_hat = H, the true channel.

Parameters

z [Input N_OFDM x Nf 2D array containing pilot blocks and OFDM data symbols.]

npbp [The pilot block period; if -1 use the known channel impulse response input to ht.]

alpha [The forgetting factor used to recursively estimate H_hat]

ht [The theoretical channel frquency response to allow ideal equalization provided Ncp is adequate.]

Returns

zz_out [The input z with the pilot blocks removed and one-tap equalization applied to each of the Nf carriers.]

H [The channel estimate in the frequency domain; an array of length Nf; will return Ht if provided as an input.]

Examples

```
>>> from sk_dsp_comm.digitalcom import chan_est_equalize
>>> zz_out,H = chan_est_eq(z,Nf,npbp,alpha,Ht=None)
```

sk_dsp_comm.digitalcom.**eye_plot**(*x*, *l*, *s=0*)

Eye pattern plot of a baseband digital communications waveform.

The signal must be real, but can be multivalued in terms of the underlying modulation scheme. Used for BPSK eye plots in the Case Study article.

Parameters

x [ndarray of the real input data vector/array]

l [display length in samples (usually two symbols)]

s [start index]

Returns

None [A plot window opens containing the eye plot]

Notes

Increase S to eliminate filter transients.

Examples

1000 bits at 10 samples per bit with 'rc' shaping.

```python
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm import digitalcom as dc
>>> x,b, data = dc.nrz_bits(1000,10,'rc')
>>> dc.eye_plot(x,20,60)
>>> plt.show()
```
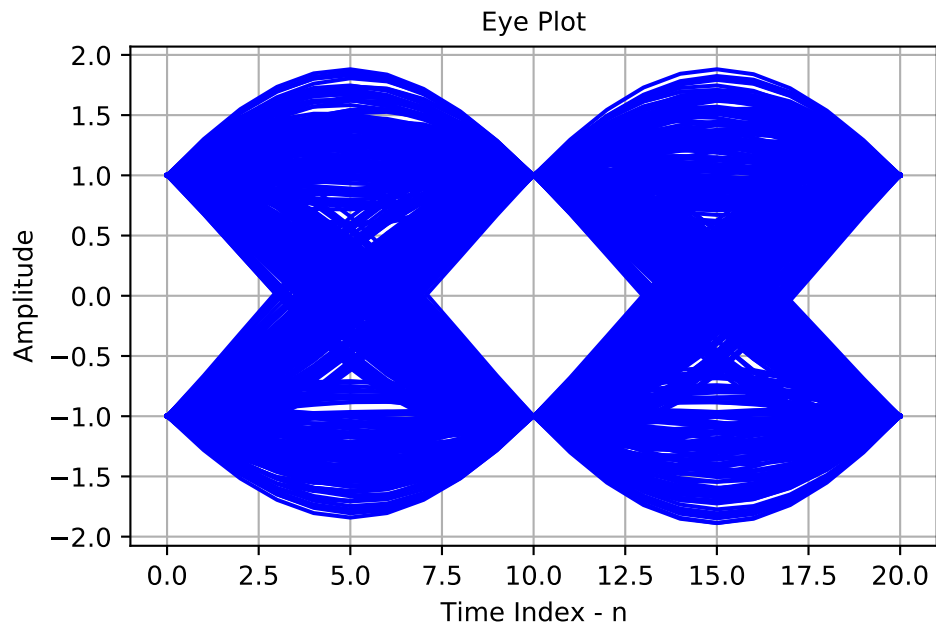

sk_dsp_comm.digitalcom.**farrow_resample**(*x*, *fs_old*, *fs_new*)

Parameters

x [Input list representing a signal vector needing resampling.]

fs_old [Starting/old sampling frequency.]

fs_new [New sampling frequency.]

Returns

> **y** [List representing the signal vector resampled at the new frequency.]

Notes

A cubic interpolator using a Farrow structure is used resample the input data at a new sampling rate that may be an irrational multiple of the input sampling rate.

Time alignment can be found for a integer value M, found with the following:

$$f_{s,out} = f_{s,in}(M-1)/M$$

The filter coefficients used here and a more comprehensive listing can be found in H. Meyr, M. Moeneclaey, & S. Fechtel, "Digital Communication Receivers," Wiley, 1998, Chapter 9, pp. 521-523.

Another good paper on variable interpolators is: L. Erup, F. Gardner, & R. Harris, "Interpolation in Digital Modems–Part II: Implementation and Performance," IEEE Comm. Trans., June 1993, pp. 998-1008.

A founding paper on the subject of interpolators is: C. W. Farrow, "A Continuously variable Digital Delay Element," Proceedings of the IEEE Intern. Symp. on Circuits Syst., pp. 2641-2645, June 1988.

Mark Wickert April 2003, recoded to Python November 2013

Examples

The following example uses a QPSK signal with rc pulse shaping, and time alignment at M = 15.

```
>>> import matplotlib.pyplot as plt
>>> from numpy import arange
>>> from sk_dsp_comm import digitalcom as dc
>>> Ns = 8
>>> Rs = 1.
>>> fsin = Ns*Rs
>>> Tsin = 1 / fsin
>>> N = 200
>>> ts = 1
>>> x, b, data = dc.mpsk_bb(N+12, Ns, 4, 'rc')
>>> x = x[12*Ns:]
>>> xxI = x.real
>>> M = 15
>>> fsout = fsin * (M-1) / M
>>> Tsout = 1. / fsout
>>> xI = dc.farrow_resample(xxI, fsin, fsin)
>>> tx = arange(0, len(xI)) / fsin
>>> yI = dc.farrow_resample(xxI, fsin, fsout)
>>> ty = arange(0, len(yI)) / fsout
>>> plt.plot(tx - Tsin, xI)
>>> plt.plot(tx[ts::Ns] - Tsin, xI[ts::Ns], 'r.')
>>> plt.plot(ty[ts::Ns] - Tsout, yI[ts::Ns], 'g.')
>>> plt.title(r'Impact of Asynchronous Sampling')
>>> plt.ylabel(r'Real Signal Amplitude')
>>> plt.xlabel(r'Symbol Rate Normalized Time')
>>> plt.xlim([0, 20])
>>> plt.grid()
>>> plt.show()
```
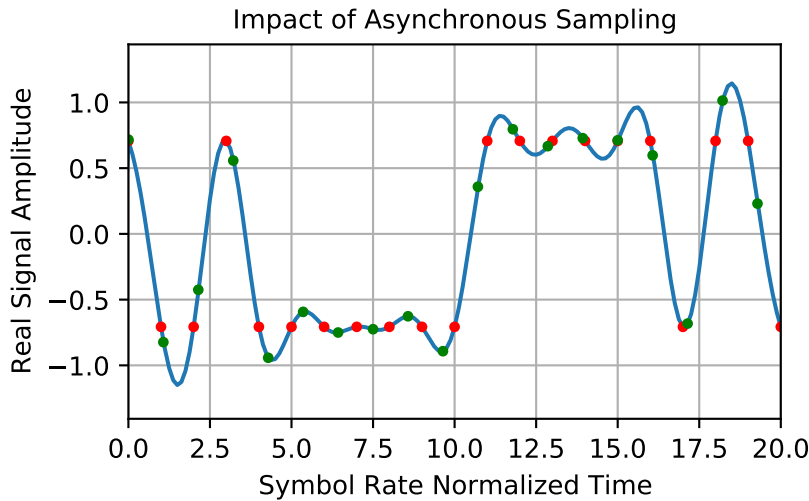
sk_dsp_comm.digitalcom.**from_bin**(*bin_array*)

> Convert binary array back a nonnegative integer. The array length is the bit width. The first input index holds the MSB and the last holds the LSB.

sk_dsp_comm.digitalcom.**gmsk_bb**(*n_bits*, *ns*, *msk=0*, *bt=0.35*)

> MSK/GMSK Complex Baseband Modulation x,data = gmsk(N_bits, Ns, BT = 0.35, MSK = 0)
>
> > **Parameters**
> >
> > > **n_bits** [number of symbols processed]
> > >
> > > **ns** [the number of samples per bit]
> > >
> > > **msk** [0 for no shaping which is standard MSK, MSK <> 0 –> GMSK is generated.]
> > >
> > > **bt** [premodulation Bb*T product which sets the bandwidth of the Gaussian lowpass filter]
> >
> > **Mark Wickert Python version November 2014**

sk_dsp_comm.digitalcom.**gray2bin**(*d_word*, *b_width*)

> Convert gray encoded binary words to integer bit words via Gray decoding starting from the MSB to the LSB
>
> Mark Wickert November 2018

sk_dsp_comm.digitalcom.**mpsk_bb**(*n_symb*, *ns*, *mod*, *pulse='rect'*, *alpha=0.25*, *m=6*)

> Generate a complex baseband MPSK signal with pulse shaping.
>
> > **Parameters**
> >
> > > **n_symb** [number of MPSK symbols to produce]
> > >
> > > **ns** [the number of samples per bit,]
> > >
> > > **mod** [MPSK modulation order, e.g., 4, 8, 16, …]
> > >
> > > **pulse** ['rect' , 'rc', 'src' (default 'rect')]
> > >
> > > **alpha** [excess bandwidth factor(default 0.25)]
> > >
> > > **m** [single sided pulse duration (default = 6)]
> >
> > **Returns**
> >
> > > **x** [ndarray of the MPSK signal values]

b [ndarray of the pulse shape]

data [ndarray of the underlying data bits]

Notes

Pulse shapes include 'rect' (rectangular), 'rc' (raised cosine), 'src' (root raised cosine). The actual pulse length is 2*M+1 samples. This function is used by BPSK_tx in the Case Study article.

Examples

```
>>> from sk_dsp_comm import digitalcom as dc
>>> import scipy.signal as signal
>>> import matplotlib.pyplot as plt
>>> x,b,data = dc.mpsk_bb(500,10,8,'src',0.35)
>>> # Matched filter received signal x
>>> y = signal.lfilter(b,1,x)
>>> plt.plot(y.real[12*10:],y.imag[12*10:])
>>> plt.xlabel('In-Phase')
>>> plt.ylabel('Quadrature')
>>> plt.axis('equal')
>>> # Sample once per symbol
>>> plt.plot(y.real[12*10::10],y.imag[12*10::10],'r.')
>>> plt.show()
```
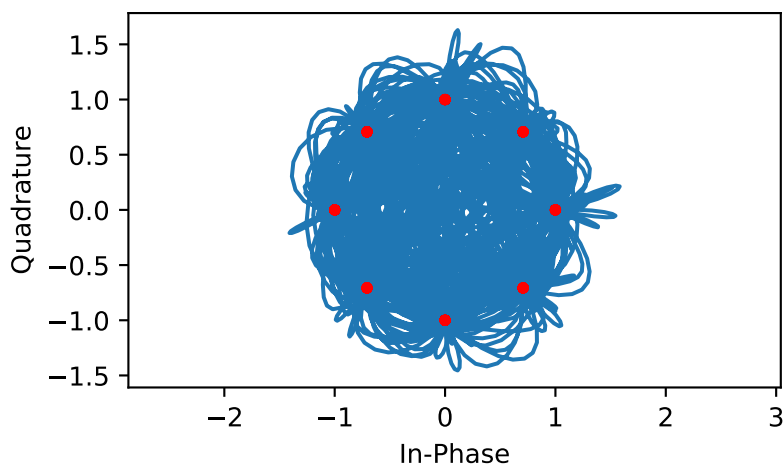

sk_dsp_comm.digitalcom.**mpsk_bep_thy**(*snr_dB*, *mod*, *eb_n0_mode=True*)
    Approximate the bit error probability of MPSK assuming Gray encoding

    Mark Wickert November 2018

sk_dsp_comm.digitalcom.**mpsk_gray_decode**(*x_hat*, *mod=4*)
    Decode MPSK IQ symbols to a serial bit stream using gray2bin decoding

        Parameters

x_hat [symbol spaced samples of the MPSK waveform taken at the maximum] eye opening. Normally this is following the matched filter

mod [Modulation scheme]

Mark Wickert November 2018

sk_dsp_comm.digitalcom.**mpsk_gray_encode_bb**(*n_symb*, *ns*, *mod=4*, *pulse='rect'*, *alpha=0.35*, *m_span=6*, *ext_data=None*)

MPSK_gray_bb: A gray code mapped MPSK complex baseband transmitter x,b,tx_data = MPSK_gray_bb(K,Ns,M)

>   **Parameters**
>
>   > **n_symb** [the number of symbols to process]
>   >
>   > **ns** [number of samples per symbol]
>   >
>   > **mod** [modulation order: 2, 4, 8, 16 MPSK]
>   >
>   > **alpha** [squareroot raised cosine excess bandwidth factor. Can range over 0 < alpha < 1.]
>   >
>   > **pulse** ['rect', 'src', or 'rc']
>
>   **Returns**
>
>   > **x** [complex baseband digital modulation]
>   >
>   > **b** [transmitter shaping filter, rectangle or SRC]
>   >
>   > **tx_data** [xI+1j*xQ = inphase symbol sequence + 1j*quadrature symbol sequence]
>   >
>   > **Mark Wickert November 2018**

sk_dsp_comm.digitalcom.**mux_pilot_blocks**(*iq_data*, *npb*)

>   **Parameters**
>
>   > **iq_data** [a 2D array of input QAM symbols with the columns] representing the NF carrier frequencies and each row the QAM symbols used to form an OFDM symbol
>   >
>   > **npb** [the period of the pilot blocks; e.g., a pilot block is] inserted every Np OFDM symbols (Np-1 OFDM data symbols of width Nf are inserted in between the pilot blocks.
>
>   **Returns**
>
>   > **IQ_datap** [IQ_data with pilot blocks inserted]
>
>   **See also:**
>
>   **OFDM_tx**

>   **Notes**
>
>   A helper function called by OFDM_tx() that inserts pilot block for use in channel estimation when a delay spread channel is present.

sk_dsp_comm.digitalcom.**my_psd**(*x*, *NFFT=1024*, *Fs=1*)

A local version of NumPy's PSD function that returns the plot arrays.

A mlab.psd wrapper function that returns two ndarrays; makes no attempt to auto plot anything.

>   **Parameters**
>
>   > **x** [ndarray input signal]

      NFFT [a power of two, e.g., 2**10 = 1024]

      Fs [the sampling rate in Hz]

   Returns

      Px [ndarray of the power spectrum estimate]

      f [ndarray of frequency values]

Notes

This function makes it easier to overlay spectrum plots because you have better control over the axis scaling than when using psd() in the autoscale mode.

Examples

```python
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm import digitalcom as dc
>>> from numpy import log10
>>> x,b, data = dc.nrz_bits(10000,10)
>>> Px,f = dc.my_psd(x,2**10,10)
>>> plt.plot(f, 10*log10(Px))
>>> plt.show()
```
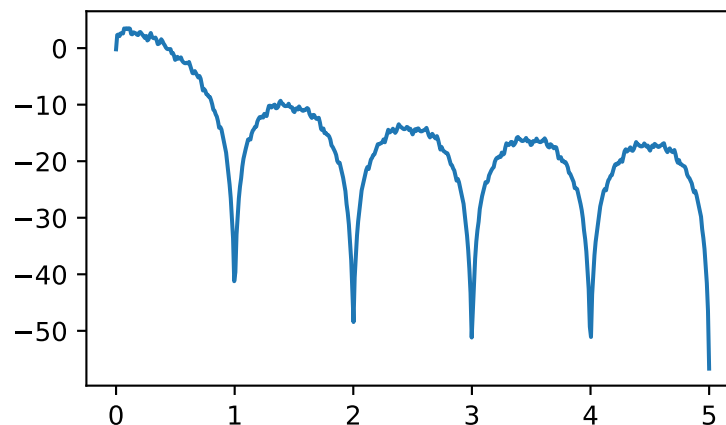

sk_dsp_comm.digitalcom.**ofdm_rx**(*x*, *nf*, *nc*, *npb=0*, *cp=False*, *ncp=0*, *alpha=0.95*, *ht=None*)

   Parameters

      x [Received complex baseband OFDM signal]

      nf [Number of filled carriers, must be even and Nf < N]

      nc [Total number of carriers; generally a power 2, e.g., 64, 1024, etc]

npb [Period of pilot code blocks; 0 <=> no pilots; -1 <=> use the ht impulse response input
to equalize the OFDM symbols; note equalization still requires Ncp > 0 to work on a delay
spread channel.]

cp [False/True <=> if False assume no CP is present]

ncp [The length of the cyclic prefix]

alpha [The filter forgetting factor in the channel estimator. Typically alpha is 0.9 to 0.99.]

ht [Input the known theoretical channel impulse response]

Returns

z_out [Recovered complex baseband QAM symbols as a serial stream; as appropriate channel
estimation has been applied.]

H [channel estimate (in the frequency domain at each subcarrier)]

See also:

`OFDM_tx`

Examples

```
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm import digitalcom as dc
>>> from scipy import signal
>>> from numpy import array
>>> hc = array([1.0, 0.1, -0.05, 0.15, 0.2, 0.05]) # impulse response spanning five
↪symbols
>>> # Quick example using the above channel with no cyclic prefix
>>> x1,b1,IQ_data1 = dc.QAM_bb(50000,1,'16qam')
>>> x_out = dc.ofdm_tx(IQ_data1,32,64,0,True,0)
>>> x1,b1,IQ_data1 = dc.qam_bb(50000,1,'16qam')
>>> x_out = dc.ofdm_tx(IQ_data1,32,64,0,True,0)
>>> c_out = signal.lfilter(hc,1,x_out) # Apply channel distortion
>>> r_out = dc.cpx_awgn(c_out,100,64/32) # Es/N0 = 100 dB
>>> z_out,H = dc.ofdm_rx(r_out,32,64,-1,True,0,alpha=0.95,ht=hc)
>>> plt.plot(z_out[200:].real,z_out[200:].imag,'.')
>>> plt.xlabel('In-Phase')
>>> plt.ylabel('Quadrature')
>>> plt.axis('equal')
>>> plt.grid()
>>> plt.show()
```

Another example with noise using a 10 symbol cyclic prefix and channel estimation:

```
>>> x_out = dc.ofdm_tx(IQ_data1,32,64,100,True,10)
>>> c_out = signal.lfilter(hc,1,x_out) # Apply channel distortion
>>> r_out = dc.cpx_awgn(c_out,25,64/32) # Es/N0 = 25 dB
>>> z_out,H = dc.ofdm_rx(r_out,32,64,100,True,10,alpha=0.95,ht=hc);
>>> plt.figure() # if channel estimation is turned on need this
>>> plt.plot(z_out[-2000:].real,z_out[-2000:].imag,'.') # allow settling time
>>> plt.xlabel('In-Phase')
>>> plt.ylabel('Quadrature')
```

(continues on next page)

```
>>> plt.axis('equal')
>>> plt.grid()
>>> plt.show()
```

sk_dsp_comm.digitalcom.**ofdm_tx**(*iq_data*, *nf*, *nc*, *npb=0*, *cp=False*, *ncp=0*)

> **Parameters**
>
> > **iq_data** [+/-1, +/-3, etc complex QAM symbol sample inputs]
> >
> > **nf** [number of filled carriers, must be even and Nf < N]
> >
> > **nc** [total number of carriers; generally a power 2, e.g., 64, 1024, etc]
> >
> > **npb** [Period of pilot code blocks; 0 <=> no pilots]
> >
> > **cp** [False/True <=> bypass cp insertion entirely if False]
> >
> > **ncp** [the length of the cyclic prefix]
>
> **Returns**
>
> > **x_out** [complex baseband OFDM waveform output after P/S and CP insertion]
>
> **See also:**
>
> **OFDM_rx**

> **Examples**

```
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm import digitalcom as dc
>>> x1,b1,IQ_data1 = dc.QAM_bb(50000,1,'16qam')
>>> x_out = dc.ofdm_tx(IQ_data1,32,64)
>>> x1,b1,IQ_data1 = dc.qam_bb(50000,1,'16qam')
>>> x_out = dc.ofdm_tx(IQ_data1,32,64)
>>> plt.psd(x_out,2**10,1);
>>> plt.xlabel(r'Normalized Frequency ($\omega/(2\pi)=f/f_s$)')
>>> plt.ylim([-40,0])
>>> plt.xlim([-.5,.5])
>>> plt.show()
```

sk_dsp_comm.digitalcom.**pcm_decode**(*x_bits*, *n_bits*)

> **Parameters**
>
> > **x_bits** [serial bit stream of 0/1 values. The length of] x_bits must be a multiple of N_bits
> >
> > **n_bits** [bit precision of PCM samples]
>
> **Returns**
>
> > **xhat** [decoded PCM signal samples]
> >
> > **Mark Wickert, March 2015**

sk_dsp_comm.digitalcom.**pcm_encode**(*x*, *n_bits*)

> **Parameters**
>
>> **x** [signal samples to be PCM encoded]
>>
>> **n_bits** [bit precision of PCM samples]
>
> **Returns**
>
>> **x_bits** [encoded serial bit stream of 0/1 values. MSB first.]
>
> **Mark Wickert, Mark 2015**

sk_dsp_comm.digitalcom.**q_fctn**(*x*)
> Gaussian Q-function

sk_dsp_comm.digitalcom.**qam_bb**(*n_symb*, *ns*, *mod='16qam'*, *pulse='rect'*, *alpha=0.35*)
> A complex baseband transmitter
>
> **Parameters**
>
>> **n_symb** [the number of symbols to process]
>>
>> **ns** [number of samples per symbol]
>>
>> **mod** [modulation type: qpsk, 16qam, 64qam, or 256qam]
>>
>> **alpha** [squareroot raised codine pulse shape bandwidth factor.] For DOCSIS alpha = 0.12 to
>> 0.18. In general alpha can range over 0 < alpha < 1.
>>
>> **pulse: pulse shapes: src, rc, rect**
>
> **Returns**
>
>> **x** [complex baseband digital modulation]
>>
>> **b** [transmitter shaping filter, rectangle or SRC]
>>
>> **tx_data** [xI+1j*xQ = inphase symbol sequence +] 1j*quadrature symbol sequence
>
> **Mark Wickert November 2014**

sk_dsp_comm.digitalcom.**qam_bep_thy**(*snr_dB*, *mod*, *eb_n0_mode=True*)
> Approximate the bit error probability of QAM assuming Gray encoding
>
> Mark Wickert November 2018

sk_dsp_comm.digitalcom.**qam_gray_decode**(*x_hat*, *mod=4*)
> Decode MQAM IQ symbols to a serial bit stream using gray2bin decoding
>
> **x_hat = symbol spaced samples of the QAM waveform taken at the maximum** eye opening. Normally this
> is following the matched filter
>
> Mark Wickert April 2018

sk_dsp_comm.digitalcom.**qam_gray_encode_bb**(*n_symb*, *ns*, *mod=4*, *pulse='rect'*, *alpha=0.35*, *m_span=6*,
ext_data=None)
> QAM_gray_bb: A gray code mapped QAM complex baseband transmitter x,b,tx_data =
> QAM_gray_bb(K,Ns,M)
>
> **Parameters**
>
>> **n_symb** [The number of symbols to process]
>>
>> **ns** [Number of samples per symbol]
>>
>> **mod** [Modulation order: 2, 4, 16, 64, 256 QAM. Note 2 <=> BPSK, 4 <=> QPSK]

> **alpha** [Square root raised cosine excess bandwidth factor.] For DOCSIS alpha = 0.12 to 0.18. In general alpha can range over 0 < alpha < 1.
>
> **pulse** ['rect', 'src', or 'rc']

> **Returns**
>
> > **x** [Complex baseband digital modulation]
> >
> > **b** [Transmitter shaping filter, rectangle or SRC]
> >
> > **tx_data** [xI+1j*xQ = inphase symbol sequence + 1j*quadrature symbol sequence]

See also:

`QAM_gray_decode`

sk_dsp_comm.digitalcom.**qam_sep**(*tx_data*, *rx_data*, *mod_type*, *Ncorr=1024*, *Ntransient=0*)

> Count symbol errors between a transmitted and received QAM signal. The received symbols are assumed to be soft values on a unit square. Time delay between streams is detected. The ndarray tx_data is Tx complex symbols. The ndarray rx_data is Rx complex symbols. Note: Ncorr needs to be even

sk_dsp_comm.digitalcom.**qpsk_bb**(*n_symb*, *ns*, *lfsr_len=5*, *pulse='src'*, *alpha=0.25*, *m=6*)

sk_dsp_comm.digitalcom.**qpsk_bep**(*tx_data*, *rx_data*, *n_corr=1024*, *n_transient=0*)

> Count bit errors between a transmitted and received QPSK signal. Time delay between streams is detected as well as ambiquity resolution due to carrier phase lock offsets of $k * \frac{\pi}{4}$, k=0,1,2,3. The ndarray sdata is Tx +/-1 symbols as complex numbers I + j*Q. The ndarray data is Rx +/-1 symbols as complex numbers I + j*Q. Note: Ncorr needs to be even

sk_dsp_comm.digitalcom.**qpsk_rx**(*fc*, *n_symb*, *rs*, *es_n0=100*, *fs=125*, *lfsr_len=10*, *phase=0*, *pulse='src'*)

> This function generates

sk_dsp_comm.digitalcom.**qpsk_tx**(*fc*, *n_symb*, *rs*, *fs=125*, *lfsr_len=10*, *pulse='src'*)

sk_dsp_comm.digitalcom.**rc_imp**(*ns*, *alpha*, *m=6*)

> A truncated raised cosine pulse used in digital communications.
>
> The pulse shaping factor $0 < \alpha < 1$ is required as well as the truncation factor M which sets the pulse duration to be $2 * M * T_{symbol}$.

> **Parameters**
>
> > **ns** [number of samples per symbol]
> >
> > **alpha** [excess bandwidth factor on (0, 1), e.g., 0.35]
> >
> > **m** [equals RC one-sided symbol truncation factor]

> **Returns**
>
> > **b** [ndarray containing the pulse shape]

See also:

[*sqrt_rc_imp*](#)

Notes

The pulse shape b is typically used as the FIR filter coefficients when forming a pulse shaped digital communications waveform.

Examples

Ten samples per symbol and $\alpha = 0.35$.

```
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm.digitalcom import rc_imp
>>> from numpy import arange
>>> b = rc_imp(10,0.35)
>>> n = arange(-10*6,10*6+1)
>>> plt.stem(n,b)
>>> plt.show()
```
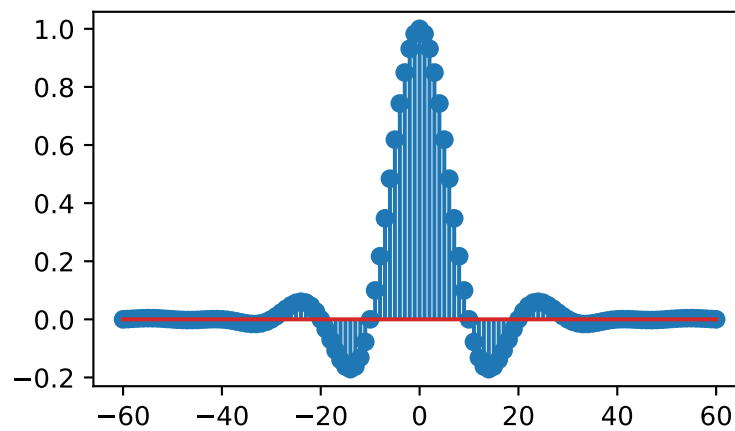

sk_dsp_comm.digitalcom.**rz_bits**(*n_bits*, *ns*, *pulse='rect'*, *alpha=0.25*, *m=6*)
Generate return-to-zero (RZ) data bits with pulse shaping.

A baseband digital data signal using +/-1 amplitude signal values and including pulse shaping.

Parameters

n_bits [number of RZ {0,1} data bits to produce]

ns [the number of samples per bit,]

pulse ['rect' , 'rc', 'src' (default 'rect')]

alpha [excess bandwidth factor(default 0.25)]

m [single sided pulse duration (default = 6)]

Returns

x [ndarray of the RZ signal values]

b [ndarray of the pulse shape]

> **data** [ndarray of the underlying data bits]

Notes

Pulse shapes include 'rect' (rectangular), 'rc' (raised cosine), 'src' (root raised cosine). The actual pulse length is 2*M+1 samples. This function is used by BPSK_tx in the Case Study article.

Examples

```
>>> import matplotlib.pyplot as plt
>>> from numpy import arange
>>> from sk_dsp_comm.digitalcom import rz_bits
>>> x,b,data = rz_bits(100,10)
>>> t = arange(len(x))
>>> plt.plot(t,x)
>>> plt.ylim([-0.01, 1.01])
>>> plt.show()
```
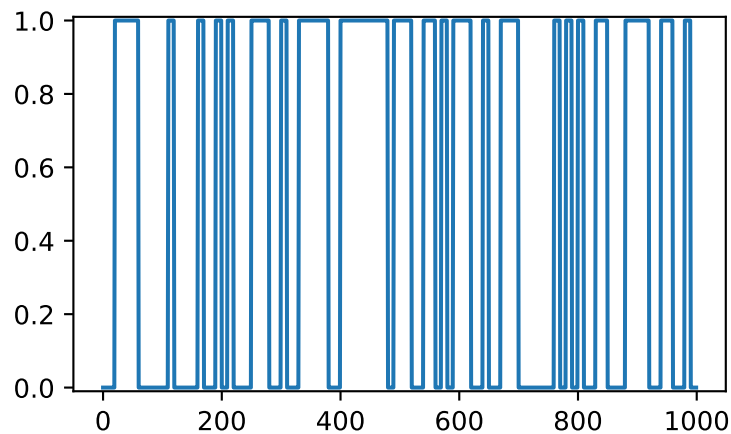

sk_dsp_comm.digitalcom.**scatter**(*x*, *ns*, *start*)

> Sample a baseband digital communications waveform at the symbol spacing.

> **Parameters**

>> **x** [ndarray of the input digital comm signal]

>> **ns** [number of samples per symbol (bit)]

>> **start** [the array index to start the sampling]

> **Returns**

>> **xI** [ndarray of the real part of x following sampling]

>> **xQ** [ndarray of the imaginary part of x following sampling]

Notes

Normally the signal is complex, so the scatter plot contains clusters at point in the complex plane. For a binary signal such as BPSK, the point centers are nominally +/-1 on the real axis. Start is used to eliminate transients from the FIR pulse shaping filters from appearing in the scatter plot.

Examples

```
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm import digitalcom as dc
>>> x,b, data = dc.nrz_bits(1000,10,'rc')
```

Add some noise so points are now scattered about +/-1.

```
>>> y = dc.cpx_awgn(x,20,10)
>>> yI,yQ = dc.scatter(y,10,60)
>>> plt.plot(yI,yQ,'.')
>>> plt.grid()
>>> plt.xlabel('In-Phase')
>>> plt.ylabel('Quadrature')
>>> plt.axis('equal')
>>> plt.show()
```
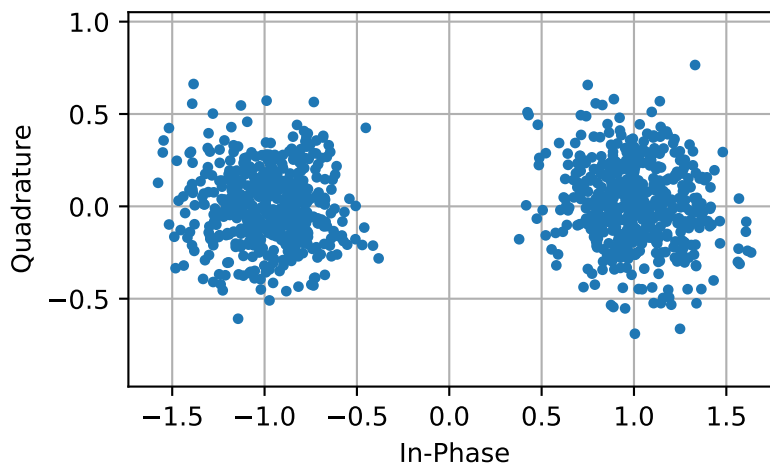

sk_dsp_comm.digitalcom.**sqrt_rc_imp**(*ns*, *alpha*, *m=6*)

A truncated square root raised cosine pulse used in digital communications.

The pulse shaping factor $0 < \alpha < 1$ is required as well as the truncation factor M which sets the pulse duration to be $2 * M * T_{symbol}$.

Parameters

ns [number of samples per symbol]

alpha [excess bandwidth factor on (0, 1), e.g., 0.35]

m [equals RC one-sided symbol truncation factor]

Returns

> **b** [ndarray containing the pulse shape]

Notes

The pulse shape b is typically used as the FIR filter coefficients when forming a pulse shaped digital communications waveform. When square root raised cosine (SRC) pulse is used to generate Tx signals and at the receiver used as a matched filter (receiver FIR filter), the received signal is now raised cosine shaped, thus having zero intersymbol interference and the optimum removal of additive white noise if present at the receiver input.

Examples

Ten samples per symbol and $\alpha = 0.35$.

```
>>> import matplotlib.pyplot as plt
>>> from numpy import arange
>>> from sk_dsp_comm.digitalcom import sqrt_rc_imp
>>> b = sqrt_rc_imp(10,0.35)
>>> n = arange(-10*6,10*6+1)
>>> plt.stem(n,b)
>>> plt.show()
```
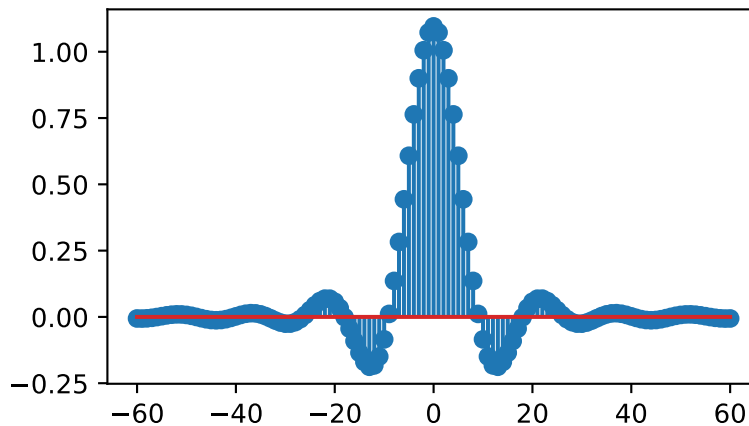
sk_dsp_comm.digitalcom.**strips**(*x, nx, fig_size=(6, 4)*)

> Plots the contents of real ndarray x as a vertical stacking of strips, each of length Nx. The default figure size is (6,4) inches. The yaxis tick labels are the starting index of each strip. The red dashed lines correspond to zero amplitude in each strip.

> strips(x,Nx,my_figsize=(6,4))

> Mark Wickert April 2014

sk_dsp_comm.digitalcom.**time_delay**(*x, d, n=4*)

> A time varying time delay which takes advantage of the Farrow structure for cubic interpolation:

> y = time_delay(x,D,N = 3)

Note that D is an array of the same length as the input signal x. This allows you to make the delay a function of time. If you want a constant delay just use D*zeros(len(x)). The minimum delay allowable is one sample or D = 1.0. This is due to the causal system nature of the Farrow structure.

A founding paper on the subject of interpolators is: C. W. Farrow, "A Continuously variable Digital Delay Element," Proceedings of the IEEE Intern. Symp. on Circuits Syst., pp. 2641-2645, June 1988.

Mark Wickert, February 2014

sk_dsp_comm.digitalcom.**to_bin**(*data*, *width*)
> Convert an unsigned integer to a numpy binary array with the first element the MSB and the last element the LSB.

sk_dsp_comm.digitalcom.**xcorr**(*x1*, *x2*, *n_lags*)
> r12, k = xcorr(x1,x2,Nlags), r12 and k are ndarray's Compute the energy normalized cross correlation between the sequences x1 and x2. If x1 = x2 the cross correlation is the autocorrelation. The number of lags sets how many lags to return centered about zero

fec_conv

A Convolutional Encoding and Decoding

Copyright (c) March 2017, Mark Wickert All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The views and conclusions contained in the software and documentation are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the FreeBSD Project.

A forward error correcting coding (FEC) class which defines methods for performing convolutional encoding and decoding. Arbitrary polynomials are supported, but the rate is presently limited to r = 1/n, where n = 2. Punctured (perforated) convolutional codes are also supported. The puncturing pattern (matrix) is arbitrary.

Two popular encoder polynomial sets are:

K = 3 ==> G1 = '111', G2 = '101' and K = 7 ==> G1 = '1011011', G2 = '1111001'.

A popular puncturing pattern to convert from rate 1/2 to rate 3/4 is a G1 output puncture pattern of '110' and a G2 output puncture pattern of '101'.

Graphical display functions are included to allow the user to better understand the operation of the Viterbi decoder.

Mark Wickert and Andrew Smit: October 2018.

class sk_dsp_comm.fec_conv.**FECConv**(*G=('111', '101')*, *Depth=10*)

> Class responsible for creating rate 1/2 convolutional code objects, and then encoding and decoding the user code set in polynomials of G. Key methods provided include *conv_encoder()*, *viterbi_decoder()*, *puncture()*, *depuncture()*, *trellis_plot()*, and *traceback_plot()*.

> > **Parameters**

> > > **G: A tuple of two binary strings corresponding to the encoder polynomials**

> > > **Depth: The decision depth employed by the Viterbi decoder method**

> ### Examples

> ```
> >>> from sk_dsp_comm import fec_conv
> >>> # Rate 1/2
> >>> cc1 = fec_conv.FECConv(('101', '111'), Depth=10)  # decision depth is 10
> ```

> ```
> >>> # Rate 1/3
> >>> from sk_dsp_comm import fec_conv
> >>> cc2 = fec_conv.FECConv(('101','011','111'), Depth=15)  # decision depth is 15
> ```

> ### Methods

> | | |
> | --- | --- |
> | *bm_calc*(ref_code_bits, rec_code_bits, ...) | distance = bm_calc(ref_code_bits, rec_code_bits, metric_type) Branch metrics calculation |
> | *conv_encoder*(input, state) | output, state = conv_encoder(input,state) We get the 1/2 or 1/3 rate from self.rate Polys G1 and G2 are entered as binary strings, e.g, G1 = '111' and G2 = '101' for K = 3 G1 = '1011011' and G2 = '1111001' for K = 7 G3 is also included for rate 1/3 Input state as a binary string of length K-1, e.g., '00' or '0000000' e.g., state = '00' for K = 3 e.g., state = '000000' for K = 7 Mark Wickert and Andrew Smit 2018 |
> | *depuncture*(soft_bits[, puncture_pattern, ...]) | Apply de-puncturing to the soft bits coming from the channel. |
> | *puncture*(code_bits[, puncture_pattern]) | Apply puncturing to the serial bits produced by convolutionally encoding. |
> | *traceback_plot*([fsize]) | Plots a path of the possible last 4 states. |
> | *trellis_plot*([fsize]) | Plots a trellis diagram of the possible state transitions. |
> | *viterbi_decoder*(x[, metric_type, quant_level]) | A method which performs Viterbi decoding of noisy bit stream, taking as input soft bit values centered on +/-1 and returning hard decision 0/1 bits. |

> **bm_calc**(*ref_code_bits*, *rec_code_bits*, *metric_type*, *quant_level*)

> > distance = bm_calc(ref_code_bits, rec_code_bits, metric_type) Branch metrics calculation

> > Mark Wickert and Andrew Smit October 2018

> **conv_encoder**(*input*, *state*)

> > output, state = conv_encoder(input,state) We get the 1/2 or 1/3 rate from self.rate Polys G1 and G2 are entered as binary strings, e.g, G1 = '111' and G2 = '101' for K = 3 G1 = '1011011' and G2 = '1111001' for K = 7 G3 is also included for rate 1/3 Input state as a binary string of length K-1, e.g., '00' or '0000000' e.g., state = '00' for K = 3 e.g., state = '000000' for K = 7 Mark Wickert and Andrew Smit 2018

depuncture(*soft_bits*, *puncture_pattern=('110', '101')*, *erase_value=3.5*)

Apply de-puncturing to the soft bits coming from the channel. Erasure bits are inserted to return the soft bit values back to a form that can be Viterbi decoded.

> **Parameters**
>
> - **soft_bits** –
> - **puncture_pattern** –
> - **erase_value** –
>
> **Returns**

Examples

This example uses the following puncture matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

The upper row operates on the outputs for the G_1 polynomial and the lower row operates on the outputs of the G_2 polynomial.

```
>>> import numpy as np
>>> from sk_dsp_comm.fec_conv import FECConv
>>> cc = FECConv(('101','111'))
>>> x = np.array([0, 0, 1, 1, 1, 0, 0, 0, 0, 0])
>>> state = '00'
>>> y, state = cc.conv_encoder(x, state)
>>> yp = cc.puncture(y, ('110','101'))
>>> cc.depuncture(yp, ('110', '101'), 1)
array([ 0., 0., 0., 1., 1., 1., 1., 0., 0., 1., 1., 0., 1., 1., 0., 1., 1., 0.]
```

puncture(*code_bits*, *puncture_pattern=('110', '101')*)

Apply puncturing to the serial bits produced by convolutionally encoding.

> **Parameters**
>
> - **code_bits** –
> - **puncture_pattern** –
>
> **Returns**

Examples

This example uses the following puncture matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

The upper row operates on the outputs for the G_1 polynomial and the lower row operates on the outputs of the G_2 polynomial.

```
>>> import numpy as np
>>> from sk_dsp_comm.fec_conv import FECConv
>>> cc = FECConv(('101','111'))
>>> x = np.array([0, 0, 1, 1, 1, 0, 0, 0, 0, 0])
>>> state = '00'
>>> y, state = cc.conv_encoder(x, state)
>>> cc.puncture(y, ('110','101'))
array([ 0.,  0.,  0.,  1.,  1.,  0.,  0.,  0.,  1.,  1.,  0.,  0.])
```

traceback_plot(*fsize=(6, 4)*)

    Plots a path of the possible last 4 states.

        Parameters

            fsize [Plot size for matplotlib.]

Examples

```
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm.fec_conv import FECConv
>>> from sk_dsp_comm import digitalcom as dc
>>> import numpy as np
>>> cc = FECConv()
>>> x = np.random.randint(0,2,100)
>>> state = '00'
>>> y,state = cc.conv_encoder(x,state)
>>> # Add channel noise to bits translated to +1/-1
>>> yn = dc.cpx_awgn(2*y-1,5,1) # SNR = 5 dB
>>> # Translate noisy +1/-1 bits to soft values on [0,7]
>>> yn = (yn.real+1)/2*7
>>> z = cc.viterbi_decoder(yn)
>>> cc.traceback_plot()
>>> plt.show()
```

trellis_plot(*fsize=(6, 4)*)

    Plots a trellis diagram of the possible state transitions.

        Parameters

            fsize [Plot size for matplotlib.]

Examples

```
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm.fec_conv import FECConv
>>> cc = FECConv()
>>> cc.trellis_plot()
>>> plt.show()
```

viterbi_decoder(*x*, *metric_type='soft'*, *quant_level=3*)

    A method which performs Viterbi decoding of noisy bit stream, taking as input soft bit values centered on +/-1 and returning hard decision 0/1 bits.

        Parameters
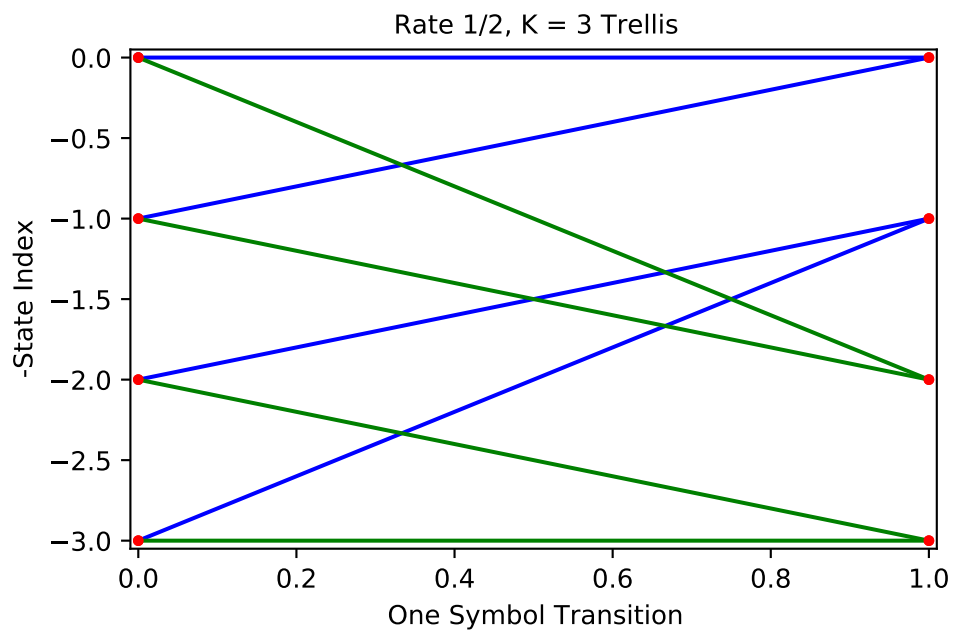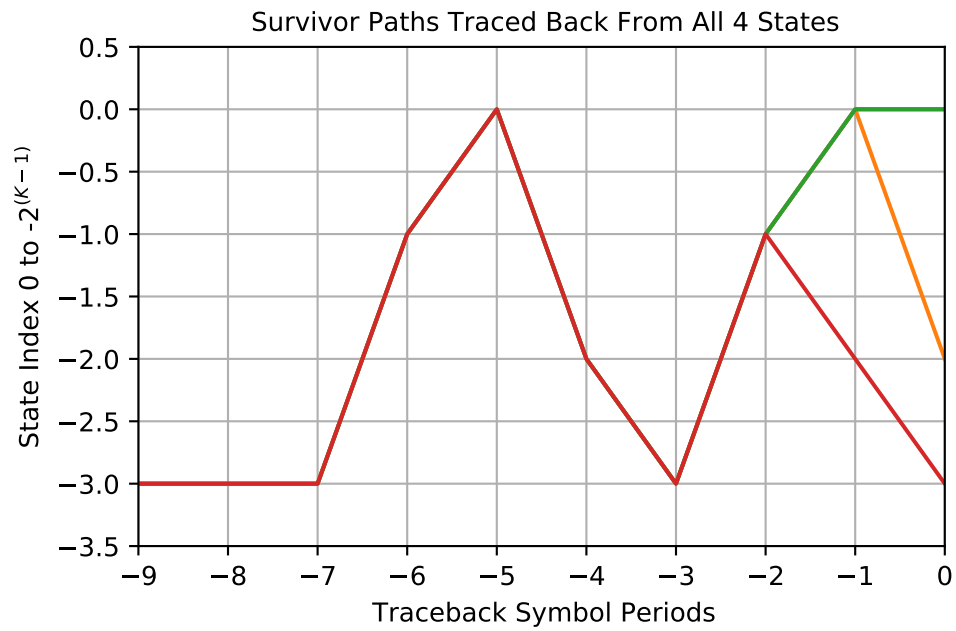
> > **x: Received noisy bit values centered on +/-1 at one sample per bit**

> > **metric_type:** 'hard' - Hard decision metric. Expects binary or 0/1 input values. 'unquant' -
> > unquantized soft decision decoding. Expects +/-1

> > > input values.

> > 'soft' - soft decision decoding.

> > **quant_level: The quantization level for soft decoding. Expected**

> > **input values between 0 and 2^quant_level-1. 0 represents the most**

> > **confident 0 and 2^quant_level-1 represents the most confident 1.**

> > **Only used for 'soft' metric type.**

> **Returns**

> > **y: Decoded 0/1 bit stream**

Examples

```python
>>> import numpy as np
>>> from numpy.random import randint
>>> import sk_dsp_comm.fec_conv as fec
>>> import sk_dsp_comm.digitalcom as dc
>>> import matplotlib.pyplot as plt
>>> # Soft decision rate 1/2 simulation
>>> N_bits_per_frame = 10000
>>> EbN0 = 4
>>> total_bit_errors = 0
>>> total_bit_count = 0
>>> cc1 = fec.FECConv(('11101','10011'),25)
>>> # Encode with shift register starting state of '0000'
>>> state = '0000'
>>> while total_bit_errors < 100:
>>>     # Create 100000 random 0/1 bits
>>>     x = randint(0,2,N_bits_per_frame)
>>>     y,state = cc1.conv_encoder(x,state)
>>>     # Add channel noise to bits, include antipodal level shift to [-1,1]
>>>     yn_soft = dc.cpx_awgn(2*y-1,EbN0-3,1) # Channel SNR is 3 dB less for
→rate 1/2
>>>     yn_hard = ((np.sign(yn_soft.real)+1)/2).astype(int)
>>>     z = cc1.viterbi_decoder(yn_hard,'hard')
>>>     # Count bit errors
>>>     bit_count, bit_errors = dc.bit_errors(x,z)
>>>     total_bit_errors += bit_errors
>>>     total_bit_count += bit_count
>>>     print('Bits Received = %d, Bit errors = %d, BEP = %1.2e' %
→     (total_bit_count, total_bit_errors,                total_bit_errors/
→total_bit_count))
>>> print('*******************************************************')
>>> print('Bits Received = %d, Bit errors = %d, BEP = %1.2e' %
→(total_bit_count, total_bit_errors,                total_bit_errors/total_bit_
→count))
Rate 1/2 Object
```

```
kmax =   0, taumax = 0
Bits Received = 9976, Bit errors = 77, BEP = 7.72e-03
kmax =   0, taumax = 0
Bits Received = 19952, Bit errors = 175, BEP = 8.77e-03
****************************************************
Bits Received = 19952, Bit errors = 175, BEP = 8.77e-03
```

```
>>> # Consider the trellis traceback after the sim completes
>>> cc1.traceback_plot()
>>> plt.show()
```
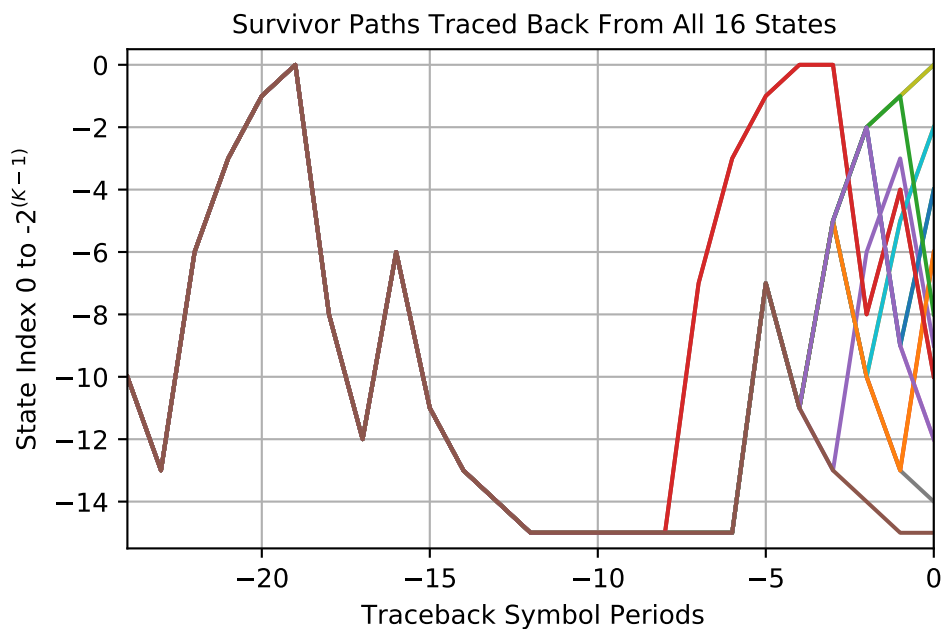


```
>>> # Compare a collection of simulation results with soft decision
>>> # bounds
>>> SNRdB = np.arange(0,12,.1)
>>> Pb_uc = fec.conv_Pb_bound(1/3,7,[4, 12, 20, 72, 225],SNRdB,2)
>>> Pb_s_third_3 = fec.conv_Pb_bound(1/3,8,[3, 0, 15],SNRdB,1)
>>> Pb_s_third_4 = fec.conv_Pb_bound(1/3,10,[6, 0, 6, 0],SNRdB,1)
>>> Pb_s_third_5 = fec.conv_Pb_bound(1/3,12,[12, 0, 12, 0, 56],SNRdB,1)
>>> Pb_s_third_6 = fec.conv_Pb_bound(1/3,13,[1, 8, 26, 20, 19, 62],SNRdB,1)
>>> Pb_s_third_7 = fec.conv_Pb_bound(1/3,14,[1, 0, 20, 0, 53, 0, 184],SNRdB,1)
>>> Pb_s_third_8 = fec.conv_Pb_bound(1/3,16,[1, 0, 24, 0, 113, 0, 287, 0],SNRdB,
→1)
>>> Pb_s_half = fec.conv_Pb_bound(1/2,7,[4, 12, 20, 72, 225],SNRdB,1)
>>> plt.figure(figsize=(5,5))
>>> plt.semilogy(SNRdB,Pb_uc)
>>> plt.semilogy(SNRdB,Pb_s_third_3,'--')
```

```
>>> plt.semilogy(SNRdB,Pb_s_third_4,'--')
>>> plt.semilogy(SNRdB,Pb_s_third_5,'g')
>>> plt.semilogy(SNRdB,Pb_s_third_6,'--')
>>> plt.semilogy(SNRdB,Pb_s_third_7,'--')
>>> plt.semilogy(SNRdB,Pb_s_third_8,'--')
>>> plt.semilogy([0,1,2,3,4,5],[9.08e-02,2.73e-02,6.52e-03,
↪         8.94e-04,8.54e-05,5e-6],'gs')
>>> plt.axis([0,12,1e-7,1e0])
>>> plt.title(r'Soft Decision Rate 1/2 Coding Measurements')
>>> plt.xlabel(r'$E_b/N_0$ (dB)')
>>> plt.ylabel(r'Symbol Error Probability')
>>> plt.legend(('Uncoded BPSK','R=1/3, K=3, Soft',                    'R=1/3,
↪K=4, Soft','R=1/3, K=5, Soft',                    'R=1/3, K=6, Soft','R=1/3,
↪K=7, Soft',                    'R=1/3, K=8, Soft','R=1/3, K=5, Sim',
↪         'Simulation'),loc='upper right')
>>> plt.grid();
>>> plt.show()
```
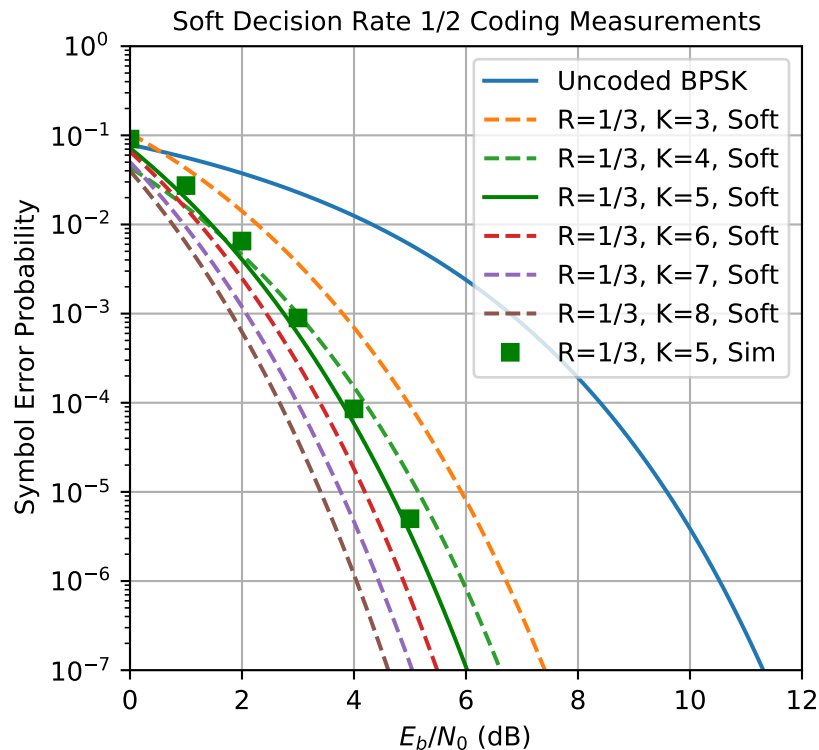


```
>>> # Hard decision rate 1/3 simulation
>>> N_bits_per_frame = 10000
```

```
>>> EbN0 = 3
>>> total_bit_errors = 0
>>> total_bit_count = 0
>>> cc2 = fec.FECConv(('11111','11011','10101'),25)
>>> # Encode with shift register starting state of '0000'
>>> state = '0000'
>>> while total_bit_errors < 100:
>>>     # Create 100000 random 0/1 bits
>>>     x = randint(0,2,N_bits_per_frame)
>>>     y,state = cc2.conv_encoder(x,state)
>>>     # Add channel noise to bits, include antipodal level shift to [-1,1]
>>>     yn_soft = dc.cpx_awgn(2*y-1,EbN0-10*np.log10(3),1) # Channel SNR is␣
→10*log10(3) dB less
>>>     yn_hard = ((np.sign(yn_soft.real)+1)/2).astype(int)
>>>     z = cc2.viterbi_decoder(yn_hard.real,'hard')
>>>     # Count bit errors
>>>     bit_count, bit_errors = dc.bit_errors(x,z)
>>>     total_bit_errors += bit_errors
>>>     total_bit_count += bit_count
>>>     print('Bits Received = %d, Bit errors = %d, BEP = %1.2e' %          ␣
→     (total_bit_count, total_bit_errors,                     total_bit_errors/
→total_bit_count))
>>> print('*****************************************************')
>>> print('Bits Received = %d, Bit errors = %d, BEP = %1.2e' %             ␣
→(total_bit_count, total_bit_errors,                   total_bit_errors/total_bit_
→count))
Rate 1/3 Object
kmax =  0, taumax = 0
Bits Received = 9976, Bit errors = 251, BEP = 2.52e-02
*****************************************************
Bits Received = 9976, Bit errors = 251, BEP = 2.52e-02
```

```
>>> # Compare a collection of simulation results with hard decision
>>> # bounds
>>> SNRdB = np.arange(0,12,.1)
>>> Pb_uc = fec.conv_Pb_bound(1/3,7,[4, 12, 20, 72, 225],SNRdB,2)
>>> Pb_s_third_3_hard = fec.conv_Pb_bound(1/3,8,[3, 0, 15, 0, 58, 0, 201, 0],
→SNRdB,0)
>>> Pb_s_third_5_hard = fec.conv_Pb_bound(1/3,12,[12, 0, 12, 0, 56, 0, 320, 0],
→SNRdB,0)
>>> Pb_s_third_7_hard = fec.conv_Pb_bound(1/3,14,[1, 0, 20, 0, 53, 0, 184],
→SNRdB,0)
>>> Pb_s_third_5_hard_sim = np.array([8.94e-04,1.11e-04,8.73e-06])
>>> plt.figure(figsize=(5,5))
>>> plt.semilogy(SNRdB,Pb_uc)
>>> plt.semilogy(SNRdB,Pb_s_third_3_hard,'r--')
>>> plt.semilogy(SNRdB,Pb_s_third_5_hard,'g--')
>>> plt.semilogy(SNRdB,Pb_s_third_7_hard,'k--')
>>> plt.semilogy(np.array([5,6,7]),Pb_s_third_5_hard_sim,'sg')
>>> plt.axis([0,12,1e-7,1e0])
>>> plt.title(r'Hard Decision Rate 1/3 Coding Measurements')
>>> plt.xlabel(r'$E_b/N_0$ (dB)')
```

```
>>> plt.ylabel(r'Symbol Error Probability')
>>> plt.legend(('Uncoded BPSK','R=1/3, K=3, Hard',                    'R=1/3,␣
→K=5, Hard', 'R=1/3, K=7, Hard',                    ),loc='upper right')
>>> plt.grid();
>>> plt.show()
```
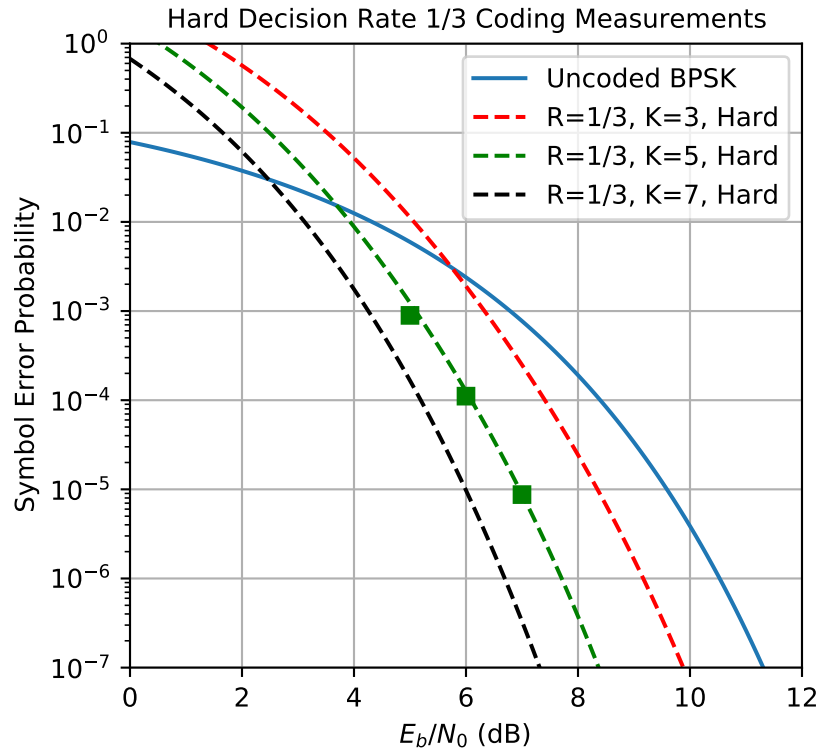


```
>>> # Show the traceback for the rate 1/3 hard decision case
>>> cc2.traceback_plot()
```

class sk_dsp_comm.fec_conv.**TrellisBranches**(*Ns*)

A structure to hold the trellis states, bits, and input values for both '1' and '0' transitions. Ns is the number of states = $2^{(K-1)}$.

class sk_dsp_comm.fec_conv.**TrellisNodes**(*Ns*)

A structure to hold the trellis from nodes and to nodes. Ns is the number of states = $2^{(K-1)}$.

class sk_dsp_comm.fec_conv.**TrellisPaths**(*Ns*, *D*)

A structure to hold the trellis paths in terms of traceback_states, cumulative_metrics, and traceback_bits. A full decision depth history of all this infomation is not essential, but does allow the graphical depiction created by the method traceback_plot(). Ns is the number of states = $2^{(K-1)}$ and D is the decision depth. As a rule, D should be about 5 times K.
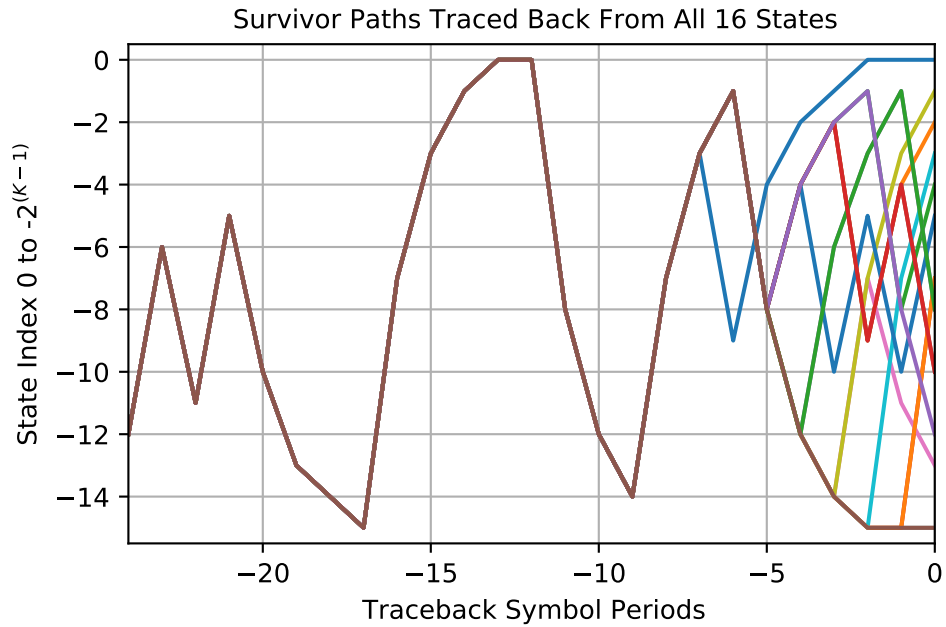
sk_dsp_comm.fec_conv.**binary**(*num*, *length=8*)

> Format an integer to binary without the leading '0b'

sk_dsp_comm.fec_conv.**conv_Pb_bound**(*R*, *dfree*, *Ck*, *SNRdB*, *hard_soft*, *M=2*)

> Coded bit error probabilty
>
> Convolution coding bit error probability upper bound according to Ziemer & Peterson 7-16, p. 507
>
> Mark Wickert November 2014

> > **Parameters**
> >
> > > **R: Code rate**
> > >
> > > **dfree: Free distance of the code**
> > >
> > > **Ck: Weight coefficient**
> > >
> > > **SNRdB: Signal to noise ratio in dB**
> > >
> > > **hard_soft: 0 hard, 1 soft, 2 uncoded**
> > >
> > > **M: M-ary**

Notes

The code rate R is given by $R_s = \frac{k}{n}$. Mark Wickert and Andrew Smit 2018

Examples

```
>>> import numpy as np
>>> from sk_dsp_comm import fec_conv as fec
>>> import matplotlib.pyplot as plt
>>> SNRdB = np.arange(2,12,.1)
>>> Pb = fec.conv_Pb_bound(1./2,10,[36, 0, 211, 0, 1404, 0, 11633],SNRdB,2)
>>> Pb_1_2 = fec.conv_Pb_bound(1./2,10,[36, 0, 211, 0, 1404, 0, 11633],SNRdB,1)
>>> Pb_3_4 = fec.conv_Pb_bound(3./4,4,[164, 0, 5200, 0, 151211, 0, 3988108],SNRdB,1)
>>> plt.semilogy(SNRdB,Pb)
>>> plt.semilogy(SNRdB,Pb_1_2)
>>> plt.semilogy(SNRdB,Pb_3_4)
>>> plt.axis([2,12,1e-7,1e0])
>>> plt.xlabel(r'$E_b/N_0$ (dB)')
>>> plt.ylabel(r'Symbol Error Probability')
>>> plt.legend(('Uncoded BPSK','R=1/2, K=7, Soft','R=3/4 (punc), K=7, Soft'),loc=
↪'best')
>>> plt.grid();
>>> plt.show()
```
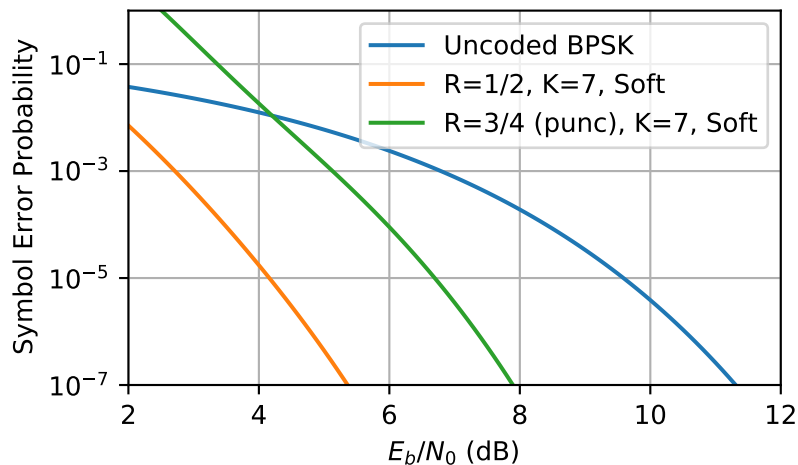

sk_dsp_comm.fec_conv.**hard_Pk**(k, R, SNR)

> Calculates Pk as found in Ziemer & Peterson eq. 7-12, p.505

> Mark Wickert and Andrew Smit 2018

sk_dsp_comm.fec_conv.**soft_Pk**(k, R, SNR)

> Calculates Pk as found in Ziemer & Peterson eq. 7-13, p.505

> Mark Wickert November 2014

fir_design_helper

Basic Linear Phase Digital Filter Design Helper

sk_dsp_comm.fir_design_helper.**bandpass_order**(*f_stop1*, *f_pass1*, *f_pass2*, *f_stop2*, *dpass_dB*, *dstop_dB*, *fsamp=1*)

Optimal FIR (equal ripple) Bandpass Order Determination

Text reference: Ifeachor, Digital Signal Processing a Practical Approach, second edition, Prentice Hall, 2002. Journal paper reference: F. Mintzer & B. Liu, Practical Design Rules for Optimum FIR Bandpass Digital Filters, IEEE Transactions on Acoustics and Speech, pp. 204-206, April,1979.

sk_dsp_comm.fir_design_helper.**bandstop_order**(*f_stop1*, *f_pass1*, *f_pass2*, *f_stop2*, *dpass_dB*, *dstop_dB*, *fsamp=1*)

Optimal FIR (equal ripple) Bandstop Order Determination

Text reference: Ifeachor, Digital Signal Processing a Practical Approach, second edition, Prentice Hall, 2002. Journal paper reference: F. Mintzer & B. Liu, Practical Design Rules for Optimum FIR Bandpass Digital Filters, IEEE Transactions on Acoustics and Speech, pp. 204-206, April,1979.

sk_dsp_comm.fir_design_helper.**fir_remez_bpf**(*f_stop1*, *f_pass1*, *f_pass2*, *f_stop2*, *d_pass*, *d_stop*, *fs=1.0*, *n_bump=5*, *status=True*)

Design an FIR bandpass filter using remez with order determination. The filter order is determined based on f_stop1 Hz, f_pass1 Hz, f_pass2 Hz, f_stop2 Hz, and the desired passband ripple d_pass dB and stopband attenuation d_stop dB all relative to a sampling rate of fs Hz.

Mark Wickert October 2016, updated October 2018

sk_dsp_comm.fir_design_helper.**fir_remez_bsf**(*f_pass1*, *f_stop1*, *f_stop2*, *f_pass2*, *d_pass*, *d_stop*, *fs=1.0*, *n_bump=5*, *status=True*)

Design an FIR bandstop filter using remez with order determination. The filter order is determined based on f_pass1 Hz, f_stop1 Hz, f_stop2 Hz, f_pass2 Hz, and the desired passband ripple d_pass dB and stopband attenuation d_stop dB all relative to a sampling rate of fs Hz.

Mark Wickert October 2016, updated October 2018

sk_dsp_comm.fir_design_helper.**fir_remez_hpf**(*f_stop*, *f_pass*, *d_pass*, *d_stop*, *fs=1.0*, *n_bump=5*, *status=True*)

> Design an FIR highpass filter using remez with order determination. The filter order is determined based on f_pass Hz, fstop Hz, and the desired passband ripple d_pass dB and stopband attenuation d_stop dB all relative to a sampling rate of fs Hz.

> Mark Wickert October 2016, updated October 2018

sk_dsp_comm.fir_design_helper.**fir_remez_lpf**(*f_pass*, *f_stop*, *d_pass*, *d_stop*, *fs=1.0*, *n_bump=5*, *status=True*)

> Design an FIR lowpass filter using remez with order determination. The filter order is determined based on f_pass Hz, fstop Hz, and the desired passband ripple d_pass dB and stopband attenuation d_stop dB all relative to a sampling rate of fs Hz.

> Mark Wickert October 2016, updated October 2018

sk_dsp_comm.fir_design_helper.**firwin_bpf**(*n_taps*, *f1*, *f2*, *fs=1.0*, *pass_zero=False*)

> Design a windowed FIR bandpass filter in terms of passband critical frequencies f1 < f2 in Hz relative to sampling rate fs in Hz. The number of taps must be provided.

> Mark Wickert October 2016

sk_dsp_comm.fir_design_helper.**firwin_kaiser_bpf**(*f_stop1*, *f_pass1*, *f_pass2*, *f_stop2*, *d_stop*, *fs=1.0*, *n_bump=0*, *status=True*)

> Design an FIR bandpass filter using the sinc() kernel and a Kaiser window. The filter order is determined based on f_stop1 Hz, f_pass1 Hz, f_pass2 Hz, f_stop2 Hz, and the desired stopband attenuation d_stop in dB for both stopbands, all relative to a sampling rate of fs Hz. Note: the passband ripple cannot be set independent of the stopband attenuation.

> Mark Wickert October 2016

sk_dsp_comm.fir_design_helper.**firwin_kaiser_bsf**(*f_stop1*, *f_pass1*, *f_pass2*, *f_stop2*, *d_stop*, *fs=1.0*, *n_bump=0*, *status=True*)

> Design an FIR bandstop filter using the sinc() kernel and a Kaiser window. The filter order is determined based on f_stop1 Hz, f_pass1 Hz, f_pass2 Hz, f_stop2 Hz, and the desired stopband attenuation d_stop in dB for both stopbands, all relative to a sampling rate of fs Hz. Note: The passband ripple cannot be set independent of the stopband attenuation. Note: The filter order is forced to be even (odd number of taps) so there is a center tap that can be used to form 1 - H_BPF.

> Mark Wickert October 2016

sk_dsp_comm.fir_design_helper.**firwin_kaiser_hpf**(*f_stop*, *f_pass*, *d_stop*, *fs=1.0*, *n_bump=0*, *status=True*)

> Design an FIR highpass filter using the sinc() kernel and a Kaiser window. The filter order is determined based on f_pass Hz, f_stop Hz, and the desired stopband attenuation d_stop in dB, all relative to a sampling rate of fs Hz. Note: the passband ripple cannot be set independent of the stopband attenuation.

> Mark Wickert October 2016

sk_dsp_comm.fir_design_helper.**firwin_kaiser_lpf**(*f_pass*, *f_stop*, *d_stop*, *fs=1.0*, *n_bump=0*, *status=True*)

> Design an FIR lowpass filter using the sinc() kernel and a Kaiser window. The filter order is determined based on f_pass Hz, f_stop Hz, and the desired stopband attenuation d_stop in dB, all relative to a sampling rate of fs Hz. Note: the passband ripple cannot be set independent of the stopband attenuation.

> Mark Wickert October 2016

sk_dsp_comm.fir_design_helper.**firwin_lpf**(*n_taps*, *fc*, *fs=1.0*)

> Design a windowed FIR lowpass filter in terms of passband critical frequencies f1 < f2 in Hz relative to sampling rate fs in Hz. The number of taps must be provided.

Mark Wickert October 2016

sk_dsp_comm.fir_design_helper.**freqz_resp_list**(*b*, *a=array([1])*, *mode='dB'*, *fs=1.0*, *n_pts=1024*, *fsize=(6, 4)*)

A method for displaying digital filter frequency response magnitude, phase, and group delay. A plot is produced using matplotlib

freq_resp(self,mode = 'dB',Npts = 1024)

A method for displaying the filter frequency response magnitude, phase, and group delay. A plot is produced using matplotlib

freqz_resp(b,a=[1],mode = 'dB',Npts = 1024,fsize=(6,4))

b = ndarray of numerator coefficients a = ndarray of denominator coefficents

mode = display mode: 'dB' magnitude, 'phase' in radians, or 'groupdelay_s' in samples and 'groupdelay_t' in sec, all versus frequency in Hz

Npts = number of points to plot; default is 1024

fsize = figure size; defult is (6,4) inches

Mark Wickert, January 2015

sk_dsp_comm.fir_design_helper.**lowpass_order**(*f_pass*, *f_stop*, *dpass_dB*, *dstop_dB*, *fsamp=1*)
Optimal FIR (equal ripple) Lowpass Order Determination

Text reference: Ifeachor, Digital Signal Processing a Practical Approach, second edition, Prentice Hall, 2002. Journal paper reference: Herriman et al., Practical Design Rules for Optimum Finite Imulse Response Digitl Filters, Bell Syst. Tech. J., vol 52, pp. 769-799, July-Aug., 1973.IEEE, 1973.

iir_design_helper

Basic IIR Bilinear Transform-Based Digital Filter Design Helper

sk_dsp_comm.iir_design_helper.**IIR_bpf**(*f_stop1*, *f_pass1*, *f_pass2*, *f_stop2*, *Ripple_pass*, *Atten_stop*, *fs=1.0*, *ftype='butter'*, *status=True*)

> Design an IIR bandpass filter using scipy.signal.iirdesign. The filter order is determined based on f_pass Hz, f_stop Hz, and the desired stopband attenuation d_stop in dB, all relative to a sampling rate of fs Hz.

> > **Parameters**

> > > **f_stop1** [ndarray of the numerator coefficients]

> > > **f_pass** [ndarray of the denominator coefficients]

> > > **Ripple_pass :**

> > > **Atten_stop :**

> > > **fs** [sampling rate in Hz]

> > > **ftype** [Analog prototype from 'butter' 'cheby1', 'cheby2',] 'ellip', and 'bessel'

> > **Returns**

> > > **b** [ndarray of the numerator coefficients]

> > > **a** [ndarray of the denominator coefficients]

> > > **sos** [2D ndarray of second-order section coefficients]

> **Examples**

```
>>> fs = 48000
>>> f_pass = 8000
>>> f_stop = 5000
>>> b_but,a_but,sos_but = IIR_hpf(f_stop,f_pass,0.5,60,fs,'butter')
>>> b_cheb1,a_cheb1,sos_cheb1 = IIR_hpf(f_stop,f_pass,0.5,60,fs,'cheby1')
>>> b_cheb2,a_cheb2,sos_cheb2 = IIR_hpf(f_stop,f_pass,0.5,60,fs,'cheby2')
>>> b_elli,a_elli,sos_elli = IIR_hpf(f_stop,f_pass,0.5,60,fs,'ellip')
```

> Mark Wickert October 2016

sk_dsp_comm.iir_design_helper.**IIR_bsf**(*f_pass1*, *f_stop1*, *f_stop2*, *f_pass2*, *Ripple_pass*, *Atten_stop*, *fs=1.0*, *ftype='butter'*, *status=True*)

> Design an IIR bandstop filter using scipy.signal.iirdesign. The filter order is determined based on f_pass Hz, f_stop Hz, and the desired stopband attenuation d_stop in dB, all relative to a sampling rate of fs Hz.

> Mark Wickert October 2016

sk_dsp_comm.iir_design_helper.**IIR_hpf**(*f_stop*, *f_pass*, *Ripple_pass*, *Atten_stop*, *fs=1.0*, *ftype='butter'*, *status=True*)

> Design an IIR highpass filter using scipy.signal.iirdesign. The filter order is determined based on f_pass Hz, f_stop Hz, and the desired stopband attenuation d_stop in dB, all relative to a sampling rate of fs Hz.

> > **Parameters**

> > > **f_stop :**

> > > **f_pass :**

> > > **Ripple_pass :**

> > > **Atten_stop :**

> > > **fs** [sampling rate in Hz]

> **ftype** [Analog prototype from 'butter' 'cheby1', 'cheby2',] 'ellip', and 'bessel'

> **Returns**

>> **b** [ndarray of the numerator coefficients]

>> **a** [ndarray of the denominator coefficients]

>> **sos** [2D ndarray of second-order section coefficients]

Examples

```
>>> fs = 48000
>>> f_pass = 8000
>>> f_stop = 5000
>>> b_but,a_but,sos_but = IIR_hpf(f_stop,f_pass,0.5,60,fs,'butter')
>>> b_cheb1,a_cheb1,sos_cheb1 = IIR_hpf(f_stop,f_pass,0.5,60,fs,'cheby1')
>>> b_cheb2,a_cheb2,sos_cheb2 = IIR_hpf(f_stop,f_pass,0.5,60,fs,'cheby2')
>>> b_elli,a_elli,sos_elli = IIR_hpf(f_stop,f_pass,0.5,60,fs,'ellip')
```

Mark Wickert October 2016

sk_dsp_comm.iir_design_helper.**IIR_lpf**(*f_pass, f_stop, Ripple_pass, Atten_stop, fs=1.0, ftype='butter', status=True*)

Design an IIR lowpass filter using scipy.signal.iirdesign. The filter order is determined based on f_pass Hz, f_stop Hz, and the desired stopband attenuation d_stop in dB, all relative to a sampling rate of fs Hz.

> **Parameters**

>> **f_pass** [Passband critical frequency in Hz]

>> **f_stop** [Stopband critical frequency in Hz]

>> **Ripple_pass** [Filter gain in dB at f_pass]

>> **Atten_stop** [Filter attenuation in dB at f_stop]

>> **fs** [Sampling rate in Hz]

>> **ftype** [Analog prototype from 'butter' 'cheby1', 'cheby2',] 'ellip', and 'bessel'

> **Returns**

>> **b** [ndarray of the numerator coefficients]

>> **a** [ndarray of the denominator coefficients]

>> **sos** [2D ndarray of second-order section coefficients]

Notes

Additionally a text string telling the user the filter order is written to the console, e.g., IIR cheby1 order = 8.

Examples

```
>>> fs = 48000
>>> f_pass = 5000
>>> f_stop = 8000
>>> b_but,a_but,sos_but = IIR_lpf(f_pass,f_stop,0.5,60,fs,'butter')
>>> b_cheb1,a_cheb1,sos_cheb1 = IIR_lpf(f_pass,f_stop,0.5,60,fs,'cheby1')
>>> b_cheb2,a_cheb2,sos_cheb2 = IIR_lpf(f_pass,f_stop,0.5,60,fs,'cheby2')
>>> b_elli,a_elli,sos_elli = IIR_lpf(f_pass,f_stop,0.5,60,fs,'ellip')
```

Mark Wickert October 2016

sk_dsp_comm.iir_design_helper.**freqz_cas**(*sos*, *w*)

> Cascade frequency response

> Mark Wickert October 2016

sk_dsp_comm.iir_design_helper.**freqz_resp_cas_list**(*sos*, *mode='dB'*, *fs=1.0*, *n_pts=1024*, *fsize=(6, 4)*)

> A method for displaying cascade digital filter form frequency response magnitude, phase, and group delay. A plot is produced using matplotlib

> freq_resp(self,mode = 'dB',Npts = 1024)

> A method for displaying the filter frequency response magnitude, phase, and group delay. A plot is produced using matplotlib

> freqz_resp(b,a=[1],mode = 'dB',Npts = 1024,fsize=(6,4))

>> b = ndarray of numerator coefficients a = ndarray of denominator coefficents

>> **mode = display mode: 'dB' magnitude, 'phase' in radians, or** 'groupdelay_s' in samples and 'groupdelay_t' in sec, all versus frequency in Hz

>> Npts = number of points to plot; default is 1024

> fsize = figure size; defult is (6,4) inches

> Mark Wickert, January 2015

sk_dsp_comm.iir_design_helper.**freqz_resp_list**(*b*, *a=array([1])*, *mode='dB'*, *fs=1.0*, *Npts=1024*, *fsize=(6, 4)*)

> A method for displaying digital filter frequency response magnitude, phase, and group delay. A plot is produced using matplotlib

> freq_resp(self,mode = 'dB',Npts = 1024)

> A method for displaying the filter frequency response magnitude, phase, and group delay. A plot is produced using matplotlib

> freqz_resp(b,a=[1],mode = 'dB',Npts = 1024,fsize=(6,4))

>> b = ndarray of numerator coefficients a = ndarray of denominator coefficents

>> **mode = display mode: 'dB' magnitude, 'phase' in radians, or** 'groupdelay_s' in samples and 'groupdelay_t' in sec, all versus frequency in Hz

>> Npts = number of points to plot; default is 1024

> fsize = figure size; defult is (6,4) inches

> Mark Wickert, January 2015

sk_dsp_comm.iir_design_helper.**sos_cascade**(*sos1*, *sos2*)
  Mark Wickert October 2016

sk_dsp_comm.iir_design_helper.**sos_zplane**(*sos*, *auto_scale=True*, *size=2*, *tol=0.001*)
  Create an z-plane pole-zero plot.

  Create an z-plane pole-zero plot using the numerator and denominator z-domain system function coefficient ndarrays b and a respectively. Assume descending powers of z.

> **Parameters**
>> **sos** [ndarray of the sos coefficients]
>>
>> **auto_scale** [bool (default True)]
>>
>> **size** [plot radius maximum when scale = False]
>
> **Returns**
>> **(M,N)** [tuple of zero and pole counts + plot window]

> **Notes**
>
> This function tries to identify repeated poles and zeros and will place the multiplicity number above and to the right of the pole or zero. The difficulty is setting the tolerance for this detection. Currently it is set at 1e-3 via the function signal.unique_roots.

> **Examples**

```
>>> # Here the plot is generated using auto_scale
>>> sos_zplane(sos)
>>> # Here the plot is generated using manual scaling
>>> sos_zplane(sos,False,1.5)
```

sk_dsp_comm.iir_design_helper.**unique_cpx_roots**(*rlist*, *tol=0.001*)
  The average of the root values is used when multiplicity is greater than one.

  Mark Wickert October 2016

multirate_helper

Multirate help for building interpolation and decimation systems

Copyright (c) March 2017, Mark Wickert All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT

SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCI-
DENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSI-
NESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CON-
TRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The views and conclusions contained in the software and documentation are those of the authors and should not be
interpreted as representing official policies, either expressed or implied, of the FreeBSD Project.

sk_dsp_comm.multirate_helper.**freqz_resp**(*b*, *a=[1]*, *mode='dB'*, *fs=1.0*, *Npts=1024*, *fsize=(6, 4)*)
> A method for displaying digital filter frequency response magnitude, phase, and group delay. A plot is produced
> using matplotlib

> freq_resp(self,mode = 'dB',Npts = 1024)

> A method for displaying the filter frequency response magnitude, phase, and group delay. A plot is produced
> using matplotlib

> freqz_resp(b,a=[1],mode = 'dB',Npts = 1024,fsize=(6,4))

>> b = ndarray of numerator coefficients a = ndarray of denominator coefficents

>> **mode = display mode: 'dB' magnitude, 'phase' in radians, or** 'groupdelay_s' in samples and
>> 'groupdelay_t' in sec, all versus frequency in Hz

>> Npts = number of points to plot; defult is 1024

> fsize = figure size; defult is (6,4) inches

> Mark Wickert, January 2015

class sk_dsp_comm.multirate_helper.**multirate_FIR**(*b*)
> A simple class for encapsulating FIR filtering, or FIR upsample/ filter, or FIR filter/downsample operations used
> in modeling a comm system. Objects of this class will hold the required filter coefficients once an object is
> instantiated. Frequency response and the pole zero plot can also be plotted using supplied class methods.

> Mark Wickert March 2017

Methods

dn(x[, M_change])	Downsample and filter the signal
filter(x)	Filter the signal
up(x[, L_change])	Upsample and filter the signal
zplane([auto_scale, size, detect_mult, tol])	Plot the poles and zeros of the FIR filter in the z-plane

freq_resp	

dn(*x*, *M_change=12*)
> Downsample and filter the signal

filter(*x*)
> Filter the signal

freq_resp(*mode='dB'*, *fs=8000*, *ylim=[- 100, 2]*)

up(*x*, *L_change=12*)
    Upsample and filter the signal

zplane(*auto_scale=True*, *size=2*, *detect_mult=True*, *tol=0.001*)
    Plot the poles and zeros of the FIR filter in the z-plane

class sk_dsp_comm.multirate_helper.**multirate_IIR**(*sos*)
    A simple class for encapsulating IIR filtering, or IIR upsample/ filter, or IIR filter/downsample operations used in modeling a comm system. Objects of this class will hold the required filter coefficients once an object is instantiated. Frequency response and the pole zero plot can also be plotted using supplied class methods. For added robustness to floating point quantization all filtering is done using the scipy.signal cascade of second-order sections filter method y = sosfilter(sos,x).

    Mark Wickert March 2017

    ### Methods

dn(x[, M_change])	Downsample and filter the signal
filter(x)	Filter the signal using second-order sections
freq_resp([mode, fs, ylim])	Frequency response plot
up(x[, L_change])	Upsample and filter the signal
zplane([auto_scale, size, detect_mult, tol])	Plot the poles and zeros of the FIR filter in the z-plane

    dn(*x*, *M_change=12*)
        Downsample and filter the signal

    filter(*x*)
        Filter the signal using second-order sections

    freq_resp(*mode='dB'*, *fs=8000*, *ylim=[- 100, 2]*)
        Frequency response plot

    up(*x*, *L_change=12*)
        Upsample and filter the signal

    zplane(*auto_scale=True*, *size=2*, *detect_mult=True*, *tol=0.001*)
        Plot the poles and zeros of the FIR filter in the z-plane

class sk_dsp_comm.multirate_helper.**rate_change**(*M_change=12*, *fcutoff=0.9*, *N_filt_order=8*, *ftype='butter'*)
    A simple class for encapsulating the upsample/filter and filter/downsample operations used in modeling a comm system. Objects of this class will hold the required filter coefficients once an object is instantiated.

    Mark Wickert February 2015

    ### Methods

dn(x)	Downsample and filter the signal
up(x)	Upsample and filter the signal

    dn(*x*)
        Downsample and filter the signal

    up(*x*)
        Upsample and filter the signal

sigsys

Signals and Systems Function Module

Copyright (c) March 2017, Mark Wickert All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The views and conclusions contained in the software and documentation are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the FreeBSD Project.

Notes

The primary purpose of this function library is to support the book Signals and Systems for Dummies. Beyond that it should be useful to anyone who wants to use Pylab for general signals and systems modeling and simulation. There is a good collection of digital communication simulation primitives included in the library. More enhancements are planned over time.

The formatted docstrings for the library follow. Click index in the upper right to get an alphabetical listing of the library functions. In all of the example code given it is assumed that ssd has been imported into your workspace. See the examples below for import options.

Examples

```
>>> import sk_dsp_comm.sigsys as ssd
>>> # Commands then need to be prefixed with ssd., i.e.,
>>> ssd.tri(t,tau)
>>> # A full import of the module, to avoid the the need to prefix with ssd, is:
>>> from sk_dsp_comm.sigsys import *
```

Function Catalog

sk_dsp_comm.sigsys.**am_rx**(*x192*)

> AM envelope detector receiver for the Chapter 17 Case Study
>
> The receiver bandpass filter is not included in this function.
>
> > **Parameters**
> >
> > > **x192** [ndarray of the AM signal at sampling rate 192 ksps]
> >
> > **Returns**
> >
> > > **m_rx8** [ndarray of the demodulated message at 8 ksps]
> > >
> > > **t8** [ndarray of the time axis at 8 ksps]
> > >
> > > **m_rx192** [ndarray of the demodulated output at 192 ksps]
> > >
> > > **x_edet192** [ndarray of the envelope detector output at 192 ksps]
>
> ### Notes
>
> The bandpass filter needed at the receiver front-end can be designed using b_bpf,a_bpf = am_rx_BPF().
>
> ### Examples
>
> ```
> >>> import numpy as np
> >>> n = np.arange(0,1000)
> >>> # 1 kHz message signal
> >>> m = np.cos(2*np.pi*1000/8000.*n)
> >>> m_rx8,t8,m_rx192,x_edet192 = am_rx(x192)
> ```

sk_dsp_comm.sigsys.**am_rx_bpf**(*n_order=7*, *ripple_dB=1*, *b=10000.0*, *fs=192000.0*)

> Bandpass filter design for the AM receiver Case Study of Chapter 17.
>
> Design a 7th-order Chebyshev type 1 bandpass filter to remove/reduce adjacent channel intereference at the envelope detector input.
>
> > **Parameters**
> >
> > > **n_order** [the filter order (default = 7)]
> > >
> > > **ripple_dB** [the passband ripple in dB (default = 1)]
> > >
> > > **b** [the RF bandwidth (default = 10e3)]
> > >
> > > **fs** [the sampling frequency]
> >
> > **Returns**
> >
> > > **b_bpf** [ndarray of the numerator filter coefficients]
> > >
> > > **a_bpf** [ndarray of the denominator filter coefficients]

Examples

```
>>> from scipy import signal
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import sk_dsp_comm.sigsys as ss
>>> # Use the default values
>>> b_bpf,a_bpf = ss.am_rx_bpf()
```

Pole-zero plot of the filter.

```
>>> ss.zplane(b_bpf,a_bpf)
>>> plt.show()
```
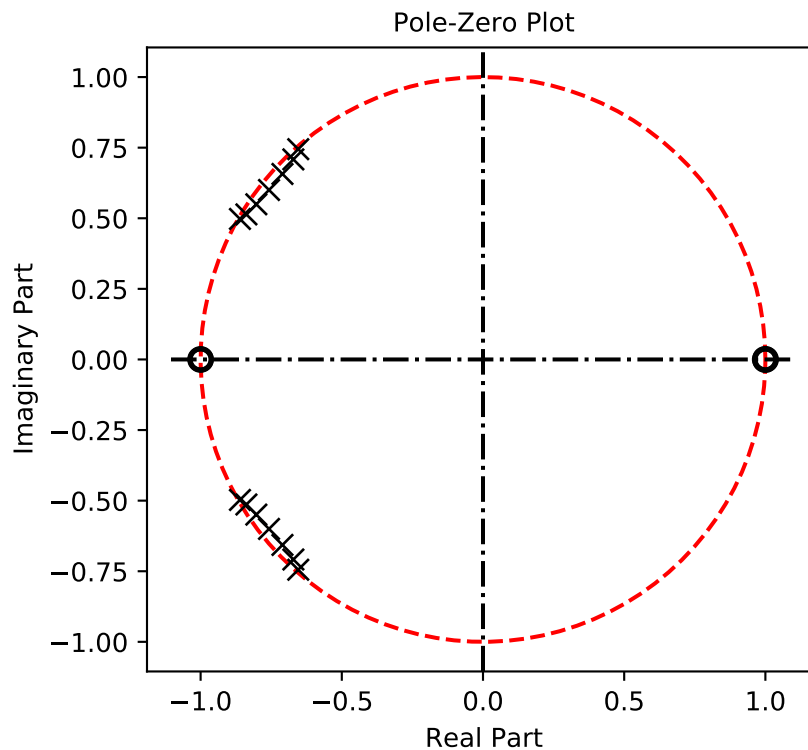

Plot of the frequency response.

```
>>> f = np.arange(0,192/2.,.1)
>>> w, Hbpf = signal.freqz(b_bpf,a_bpf,2*np.pi*f/192)
>>> plt.plot(f*10,20*np.log10(abs(Hbpf)))
>>> plt.axis([0,1920/2.,-80,10])
>>> plt.ylabel("Power Spectral Density (dB)")
>>> plt.xlabel("Frequency (kHz)")
```

```
>>> plt.show()
```
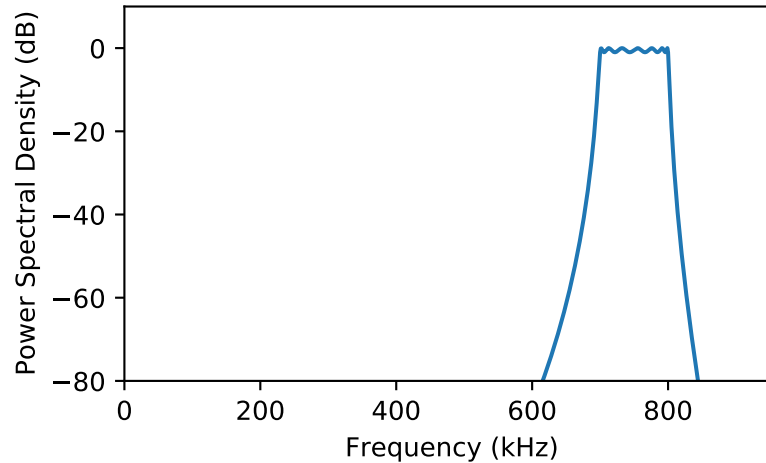

sk_dsp_comm.sigsys.**am_tx**(*m*, *a_mod*, *fc=75000.0*)

AM transmitter for Case Study of Chapter 17.

Assume input is sampled at 8 Ksps and upsampling by 24 is performed to arrive at fs_out = 192 Ksps.

Parameters

m [ndarray of the input message signal]

a_mod [AM modulation index, between 0 and 1]

fc [the carrier frequency in Hz]

Returns

x192 [ndarray of the upsampled by 24 and modulated carrier]

t192 [ndarray of the upsampled by 24 time axis]

m24 [ndarray of the upsampled by 24 message signal]

Notes

The sampling rate of the input signal is assumed to be 8 kHz.

Examples

```
>>> n = arange(0,1000)
>>> # 1 kHz message signal
>>> m = cos(2*pi*1000/8000.*n)
>>> x192, t192 = am_tx(m,0.8,fc=75e3)
```

sk_dsp_comm.sigsys.**bin_num**(*n*, *n_bits*)

    Produce a signed representation of the number n using n_bits.

        Parameters

- **n** – Number n
- **n_bits** – Number of bits

        Returns

sk_dsp_comm.sigsys.**biquad2**(*w_num*, *r_num*, *w_den*, *r_den*)

    A biquadratic filter in terms of conjugate pole and zero pairs.

        Parameters

            w_num [zero frequency (angle) in rad/sample]

            r_num [conjugate zeros radius]

            w_den [pole frequency (angle) in rad/sample]

            r_den [conjugate poles radius; less than 1 for stability]

        Returns

            b [ndarray of numerator coefficients]

            a [ndarray of denominator coefficients]

Examples

```
>>> b,a = biquad2(pi/4., 1, pi/4., 0.95)
```

sk_dsp_comm.sigsys.**bit_errors**(*z*, *data*, *start*, *ns*)

    A simple bit error counting function.

    In its present form this function counts bit errors between hard decision BPSK bits in +/-1 form and compares them with 0/1 binary data that was transmitted. Timing between the Tx and Rx data is the responsibility of the user. An enhanced version of this function, which features automatic synching will be created in the future.

        Parameters

            z [ndarray of hard decision BPSK data prior to symbol spaced sampling]

            data [ndarray of reference bits in 1/0 format]

            start [timing reference for the received]

            ns [the number of samples per symbol]

Returns

>   **Pe_hat** [the estimated probability of a bit error]

Notes

The Tx and Rx data streams are exclusive-or'd and the then the bit errors are summed, and finally divided by the number of bits observed to form an estimate of the bit error probability. This function needs to be enhanced to be more useful.

Examples

```python
>>> from scipy import signal
>>> x,b, data = nrz_bits(1000,10)
>>> # set Eb/N0 to 8 dB
>>>  y = cpx_awgn(x,8,10)
>>> # matched filter the signal
>>> z = signal.lfilter(b,1,y)
>>> # make bit decisions at 10 and Ns multiples thereafter
>>> Pe_hat = bit_errors(z,data,10,10)
```

sk_dsp_comm.sigsys.**bpsk_tx**(*N_bits*, *Ns*, *ach_fc=2.0*, *ach_lvl_dB=- 100*, *pulse='rect'*, *alpha=0.25*, *M=6*)
>   Generates biphase shift keyed (BPSK) transmitter with adjacent channel interference.

>   Generates three BPSK signals with rectangular or square root raised cosine (SRC) pulse shaping of duration N_bits and Ns samples per bit. The desired signal is centered on f = 0, which the adjacent channel signals to the left and right are also generated at dB level relative to the desired signal. Used in the digital communications Case Study supplement.

>   **Parameters**

>>   **N_bits** [the number of bits to simulate]

>>   **Ns** [the number of samples per bit]

>>   **ach_fc** [the frequency offset of the adjacent channel signals (default 2.0)]

>>   **ach_lvl_dB** [the level of the adjacent channel signals in dB (default -100)]

>>   **pulse :the pulse shape 'rect' or 'src'**

>>   **alpha** [square root raised cosine pulse shape factor (default = 0.25)]

>>   **M** [square root raised cosine pulse truncation factor (default = 6)]

>   **Returns**

>>   **x** [ndarray of the composite signal x0 + ach_lvl*(x1p + x1m)]

>>   **b** [the transmit pulse shape]

>>   **data0** [the data bits used to form the desired signal; used for error checking]

Examples

```
>>> x,b,data0 = bpsk_tx(1000,10,pulse='src')
```

sk_dsp_comm.sigsys.**cascade_filters**(*b1*, *a1*, *b2*, *a2*)

> Cascade two IIR digital filters into a single (b,a) coefficient set.
>
> To cascade two digital filters (system functions) given their numerator and denominator coefficients you simply convolve the coefficient arrays.
>
> > **Parameters**
> >
> > > **b1** [ndarray of numerator coefficients for filter 1]
> > >
> > > **a1** [ndarray of denominator coefficients for filter 1]
> > >
> > > **b2** [ndarray of numerator coefficients for filter 2]
> > >
> > > **a2** [ndarray of denominator coefficients for filter 2]
> >
> > **Returns**
> >
> > > **b** [ndarray of numerator coefficients for the cascade]
> > >
> > > **a** [ndarray of denominator coefficients for the cascade]

Examples

```
>>> from scipy import signal
>>> b1,a1 = signal.butter(3, 0.1)
>>> b2,a2 = signal.butter(3, 0.15)
>>> b,a = cascade_filters(b1,a1,b2,a2)
```

sk_dsp_comm.sigsys.**cic**(*m*, *k*)

> A functional form implementation of a cascade of integrator comb (CIC) filters.
>
> > **Parameters**
> >
> > > **m** [Effective number of taps per section (typically the decimation factor).]
> > >
> > > **k** [The number of CIC sections cascaded (larger K gives the filter a wider image rejection bandwidth).]
> >
> > **Returns**
> >
> > > **b** [FIR filter coefficients for a simple direct form implementation using the filter() function.]

Notes

Commonly used in multirate signal processing digital down-converters and digital up-converters. A true CIC filter requires no multiplies, only add and subtract operations. The functional form created here is a simple FIR requiring real coefficient multiplies via filter().

Mark Wickert July 2013

sk_dsp_comm.sigsys.**conv_integral**(*x1*, *tx1*, *x2*, *tx2*, *extent=('f', 'f')*)

> Continuous-time convolution of x1 and x2 with proper tracking of the output time axis.
>
> Approximate the convolution integral for the convolution of two continuous-time signals using the SciPy function signal. The time (sequence axis) are managed from input to output. y(t) = x1(t)*x2(t).

> **Parameters**
>
> > **x1** [ndarray of signal x1 corresponding to tx1]
> >
> > **tx1** [ndarray time axis for x1]
> >
> > **x2** [ndarray of signal x2 corresponding to tx2]
> >
> > **tx2** [ndarray time axis for x2]
> >
> > **extent** [('e1','e2') where 'e1', 'e2' may be 'f' finite, 'r' right-sided, or 'l' left-sided]
>
> **Returns**
>
> > **y** [ndarray of output values y]
> >
> > **ty** [ndarray of the corresponding time axis for y]

Notes

The output time axis starts at the sum of the starting values in x1 and x2 and ends at the sum of the two ending values in x1 and x2. The time steps used in x1(t) and x2(t) must match. The default extents of ('f','f') are used for signals that are active (have support) on or within t1 and t2 respectively. A right-sided signal such as exp(-a*t)*u(t) is semi-infinite, so it has extent 'r' and the convolution output will be truncated to display only the valid results.

Examples

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> import sk_dsp_comm.sigsys as ss
>>> tx = np.arange(-5,10,.01)
>>> x = ss.rect(tx-2,4) # pulse starts at t = 0
>>> y,ty = ss.conv_integral(x,tx,x,tx)
>>> plt.plot(ty,y) # expect a triangle on [0,8]
>>> plt.show()
```
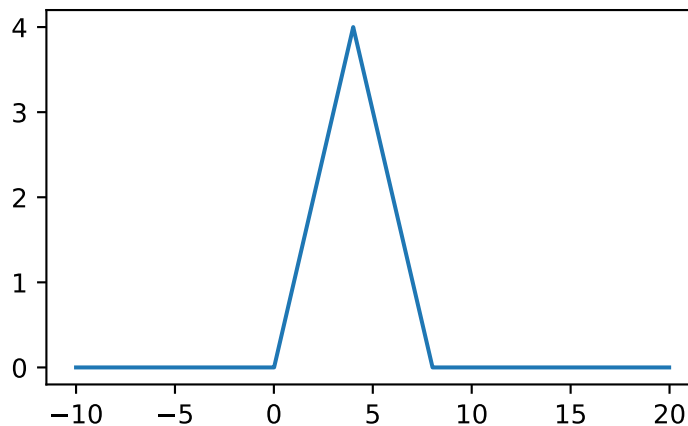
Now, consider a pulse convolved with an exponential.

```
>>> h = 4*np.exp(-4*tx)*ss.step(tx)
>>> y,ty = ss.conv_integral(x,tx,h,tx,extent=('f','r')) # note extents set
>>> plt.plot(ty,y) # expect a pulse charge and discharge waveform
```
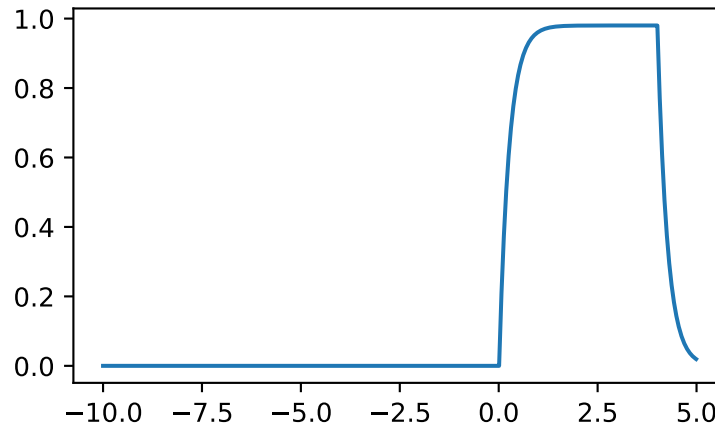

sk_dsp_comm.sigsys.**conv_sum**(*x1*, *nx1*, *x2*, *nx2*, *extent=('f', 'f')*)

> Discrete convolution of x1 and x2 with proper tracking of the output time axis.

> Convolve two discrete-time signals using the SciPy function `scipy.signal.convolution()`. The time (sequence axis) are managed from input to output. y[n] = x1[n]*x2[n].

> **Parameters**

>> **x1** [ndarray of signal x1 corresponding to nx1]

>> **nx1** [ndarray time axis for x1]

>> **x2** [ndarray of signal x2 corresponding to nx2]

>> **nx2** [ndarray time axis for x2]

>> **extent** [('e1','e2') where 'e1', 'e2' may be 'f' finite, 'r' right-sided, or 'l' left-sided]

> **Returns**

>> **y** [ndarray of output values y]

>> **ny** [ndarray of the corresponding sequence index n]

Notes

The output time axis starts at the sum of the starting values in x1 and x2 and ends at the sum of the two ending values in x1 and x2. The default extents of ('f','f') are used for signals that are active (have support) on or within n1 and n2 respectively. A right-sided signal such as a^n*u[n] is semi-infinite, so it has extent 'r' and the convolution output will be truncated to display only the valid results.

Examples

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> import sk_dsp_comm.sigsys as ss
>>> nx = np.arange(-5,10)
>>> x = ss.drect(nx,4)
>>> y,ny = ss.conv_sum(x,nx,x,nx)
>>> plt.stem(ny,y)
>>> plt.show()
```
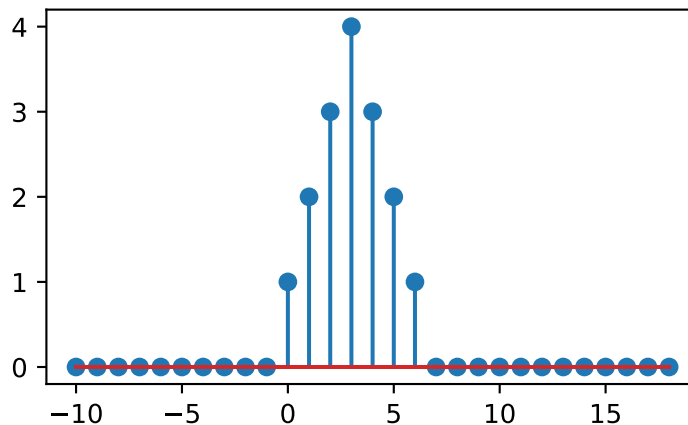

Consider a pulse convolved with an exponential. ('r' type extent)

```
>>> h = 0.5**nx*ss.dstep(nx)
>>> y,ny = ss.conv_sum(x,nx,h,nx,('f','r')) # note extents set
>>> plt.stem(ny,y) # expect a pulse charge and discharge sequence
```

sk_dsp_comm.sigsys.**cpx_awgn**(x, es_n0, ns)

Apply white Gaussian noise to a digital communications signal.

This function represents a complex baseband white Gaussian noise digital communications channel. The input signal array may be real or complex.

>**Parameters**
>
>>**x** [ndarray noise free complex baseband input signal.]
>>
>>**EsNO** [set the channel Es/N0 (Eb/N0 for binary) level in dB]
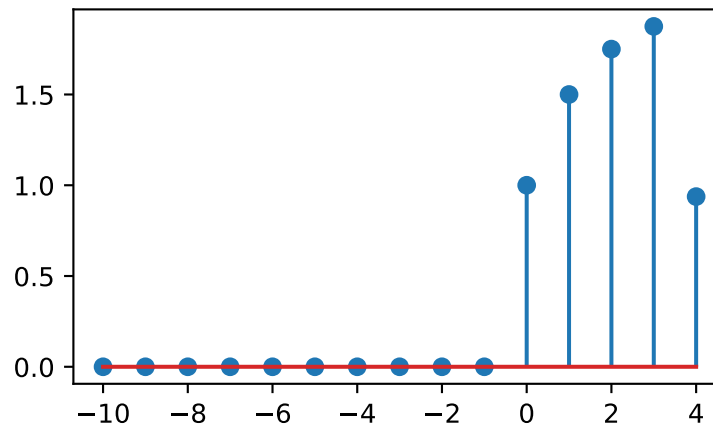
ns [number of samples per symbol (bit)]

Returns

y [ndarray x with additive noise added.]

Notes

Set the channel energy per symbol-to-noise power spectral density ratio (Es/N0) in dB.

Examples

```
>>> x,b, data = nrz_bits(1000,10)
>>> # set Eb/N0 = 10 dB
>>> y = cpx_awgn(x,10,10)
```

sk_dsp_comm.sigsys.**cruise_control**(*wn*, *zeta*, *T*, *vcruise*, *vmax*, *tf_mode='H'*)

Cruise control with PI controller and hill disturbance.

This function returns various system function configurations for a the cruise control Case Study example found in the supplementary article. The plant model is obtained by the linearizing the equations of motion and the controller contains a proportional and integral gain term set via the closed-loop parameters natural frequency wn (rad/s) and damping zeta.

Parameters

wn [closed-loop natural frequency in rad/s, nominally 0.1]

zeta [closed-loop damping factor, nominally 1.0]

T [vehicle time constant, nominally 10 s]

vcruise [cruise velocity set point, nominally 75 mph]

vmax [maximum vehicle velocity, nominally 120 mph]

tf_mode ['H', 'HE', 'HVW', or 'HED' controls the system function returned by the function]

'H' [closed-loop system function V(s)/R(s)]

'HE' [closed-loop system function E(s)/R(s)]

'HVW' [closed-loop system function V(s)/W(s)]

'HED' [closed-loop system function E(s)/D(s), where D is the hill disturbance input]

> **Returns**
>
> > **b** [numerator coefficient ndarray]
> >
> > **a** [denominator coefficient ndarray]

Examples

```
>>> # return the closed-loop system function output/input velocity
>>> b,a = cruise_control(wn,zeta,T,vcruise,vmax,tf_mode='H')
>>> # return the closed-loop system function loop error/hill disturbance
>>> b,a = cruise_control(wn,zeta,T,vcruise,vmax,tf_mode='HED')
```

sk_dsp_comm.sigsys.**deci24**(*x*)

> Decimate by L = 24 using Butterworth filters.
>
> The decimation is done using two three stages. Downsample sample by L = 2 and lowpass filter, downsample by 3 and lowpass filter, then downsample by L = 4 and lowpass filter. In all cases the lowpass filter is a 10th-order Butterworth lowpass.
>
> > **Parameters**
> >
> > > **x** [ndarray of the input signal]
> >
> > **Returns**
> >
> > > **y** [ndarray of the output signal]

Notes

The cutoff frequency of the lowpass filters is 1/2, 1/3, and 1/4 to track the upsampling by 2, 3, and 4 respectively.

Examples

```
>>> y = deci24(x)
```

sk_dsp_comm.sigsys.**delta_eps**(*t*, *eps*)

> Rectangular pulse approximation to impulse function.
>
> > **Parameters**
> >
> > > **t** [ndarray of time axis]
> > >
> > > **eps** [pulse width]
> >
> > **Returns**
> >
> > > **d** [ndarray containing the impulse approximation]

Examples

```
>>> import matplotlib.pyplot as plt
>>> from numpy import arange
>>> from sk_dsp_comm.sigsys import delta_eps
>>> t = np.arange(-2,2,.001)
>>> d = delta_eps(t,.1)
>>> plt.plot(t,d)
>>> plt.show()
```
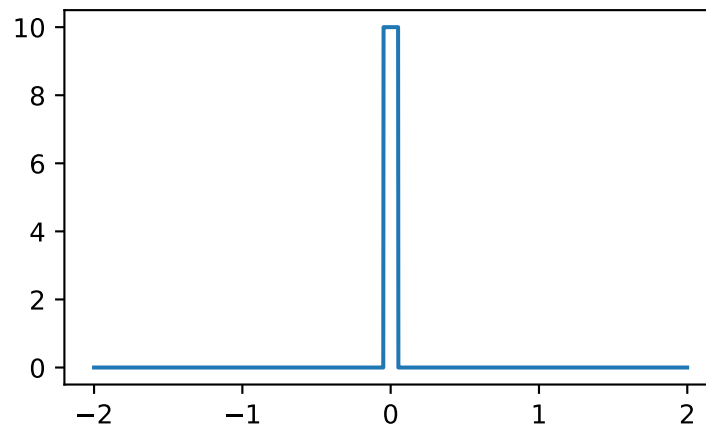
sk_dsp_comm.sigsys.**dimpulse**(*n*)

Discrete impulse function delta[n].

> **Parameters**
>
> > **n** [ndarray of the time axis]
>
> **Returns**
>
> > **x** [ndarray of the signal delta[n]]

Examples

```
>>> import matplotlib.pyplot as plt
>>> from numpy import arange
>>> from sk_dsp_comm.sigsys import dimpulse
>>> n = arange(-5,5)
>>> x = dimpulse(n)
>>> plt.stem(n,x)
>>> plt.show()
```

Shift the delta left by 2.

```
>>> x = dimpulse(n+2)
>>> plt.stem(n,x)
```
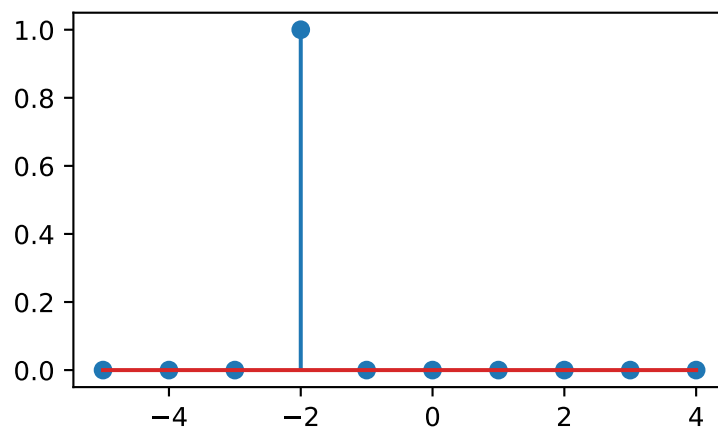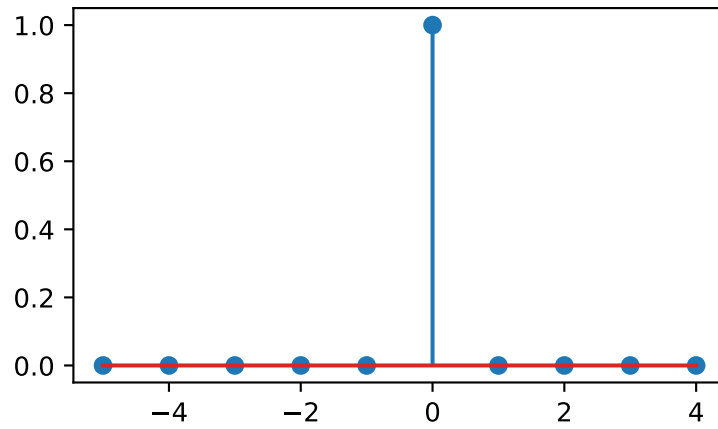
`sk_dsp_comm.sigsys.`**`downsample`**`(`*x, M, p=0*`)`

Downsample by factor M

Keep every Mth sample of the input. The phase of the input samples kept can be selected.

> **Parameters**
>
> > **x** [ndarray of input signal values]
> >
> > **M** [downsample factor]
> >
> > **p** [phase of decimated value, 0 (default), 1, ..., M-1]
>
> **Returns**
>
> > **y** [ndarray of the output signal values]

Examples

```
>>> y = downsample(x,3)
>>> y = downsample(x,3,1)
```

`sk_dsp_comm.sigsys.`**`drect`**`(`*n, N*`)`

Discrete rectangle function of duration N samples.

The signal is active on the interval 0 <= n <= N-1. Also known as the rectangular window function, which is available in scipy.signal.

> **Parameters**
>
> > **n** [ndarray of the time axis]
> >
> > **N** [the pulse duration]
>
> **Returns**
>
> > **x** [ndarray of the signal]

Notes

The discrete rectangle turns on at n = 0, off at n = N-1 and has duration of exactly N samples.

Examples

```
>>> import matplotlib.pyplot as plt
>>> from numpy import arange
>>> from sk_dsp_comm.sigsys import drect
>>> n = arange(-5,5)
>>> x = drect(n, N=3)
>>> plt.stem(n,x)
>>> plt.show()
```

Shift the delta left by 2.

```
>>> x = drect(n+2, N=3)
>>> plt.stem(n,x)
```
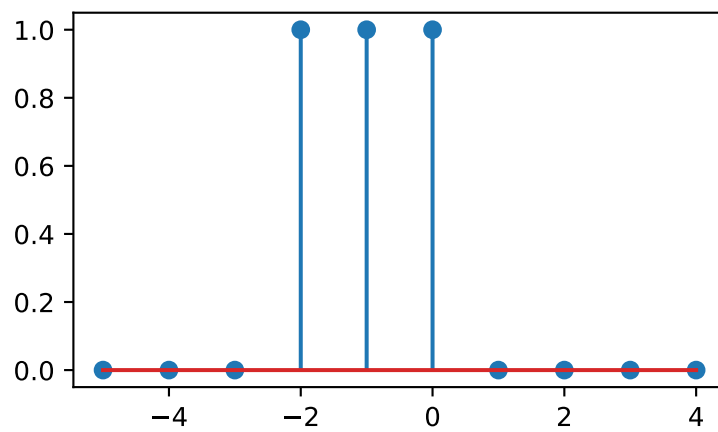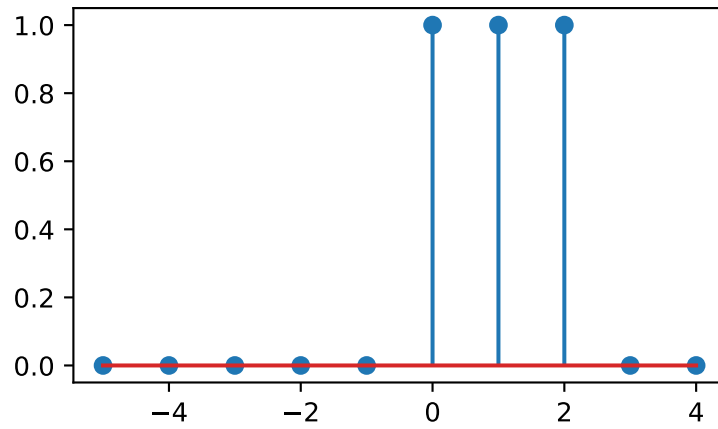
sk_dsp_comm.sigsys.**dstep**(n)

>   Discrete step function u[n].

> > **Parameters**

> > > **n** [ndarray of the time axis]

> > **Returns**

> > > **x** [ndarray of the signal u[n]]

> **Examples**

```
>>> import matplotlib.pyplot as plt
>>> from numpy import arange
>>> from sk_dsp_comm.sigsys import dstep
>>> n = arange(-5,5)
>>> x = dstep(n)
>>> plt.stem(n,x)
>>> plt.show()
```
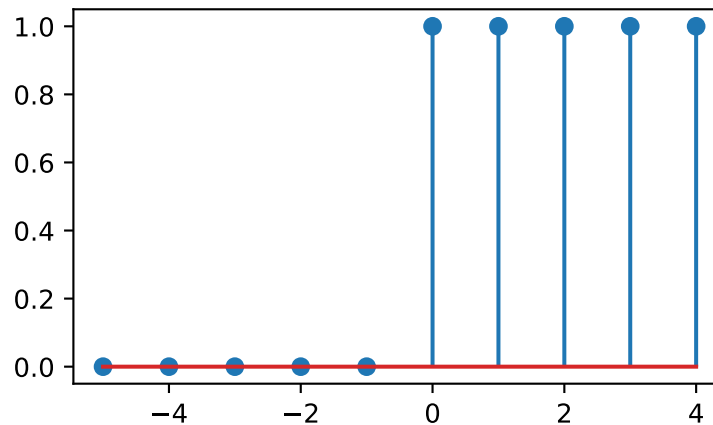

> Shift the delta left by 2.

```
>>> x = dstep(n+2)
>>> plt.stem(n,x)
```

sk_dsp_comm.sigsys.**env_det**(x)

>   Ideal envelope detector.

>   This function retains the positive half cycles of the input signal.

> > **Parameters**

> > > **x** [ndarray of the input sugnal]

> > **Returns**

> > > **y** [ndarray of the output signal]
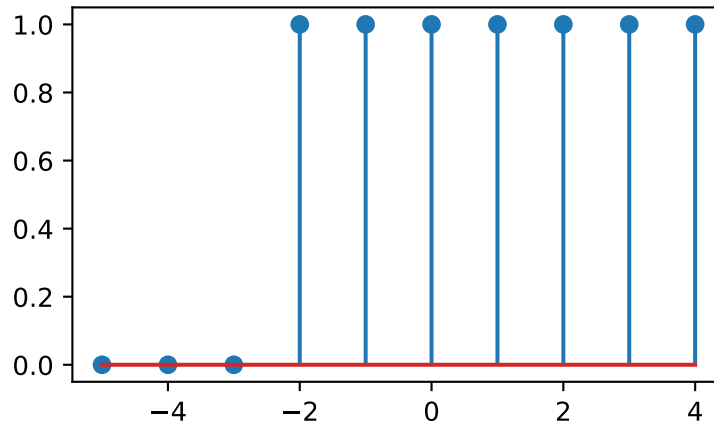
Examples

```
>>> n = arange(0,100)
>>> # 1 kHz message signal
>>> m = cos(2*pi*1000/8000.*n)
>>> x192, t192, m24 = am_tx(m,0.8,fc=75e3)
>>> y = env_det(x192)
```

`sk_dsp_comm.sigsys.ex6_2(n)`

Generate a triangle pulse as described in Example 6-2 of Chapter 6.

You need to supply an index array n that covers at least [-2, 5]. The function returns the hard-coded signal of the example.

Parameters

n [time index ndarray covering at least -2 to +5.]

Returns

x [ndarray of signal samples in x]

Examples

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm import sigsys as ss
>>> n = np.arange(-5,8)
>>> x = ss.ex6_2(n)
>>> plt.stem(n,x) # creates a stem plot of x vs n
```

`sk_dsp_comm.sigsys.eye_plot(x, l, s=0)`

Eye pattern plot of a baseband digital communications waveform.

The signal must be real, but can be multivalued in terms of the underlying modulation scheme. Used for BPSK eye plots in the Case Study article.
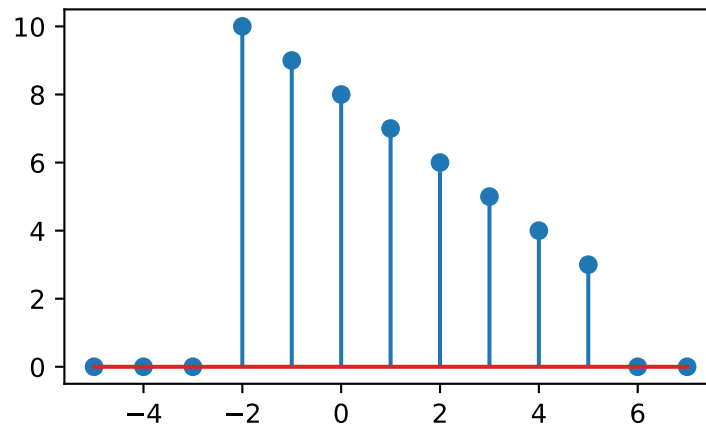
Parameters

x [ndarray of the real input data vector/array]

l [display length in samples (usually two symbols)]

s [start index]

Returns

Nothing [A plot window opens containing the eye plot]

Notes

Increase S to eliminate filter transients.

Examples

1000 bits at 10 samples per bit with 'rc' shaping.

```
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm import sigsys as ss
>>> x,b, data = ss.nrz_bits(1000,10,'rc')
>>> ss.eye_plot(x,20,60)
```

sk_dsp_comm.sigsys.**fir_iir_notch**(*fi, fs, r=0.95*)

Design a second-order FIR or IIR notch filter.

A second-order FIR notch filter is created by placing conjugate zeros on the unit circle at angle corresponidng to the notch center frequency. The IIR notch variation places a pair of conjugate poles at the same angle, but with radius r < 1 (typically 0.9 to 0.95).

Parameters

fi [notch frequency is Hz relative to fs]

fs [the sampling frequency in Hz, e.g. 8000]
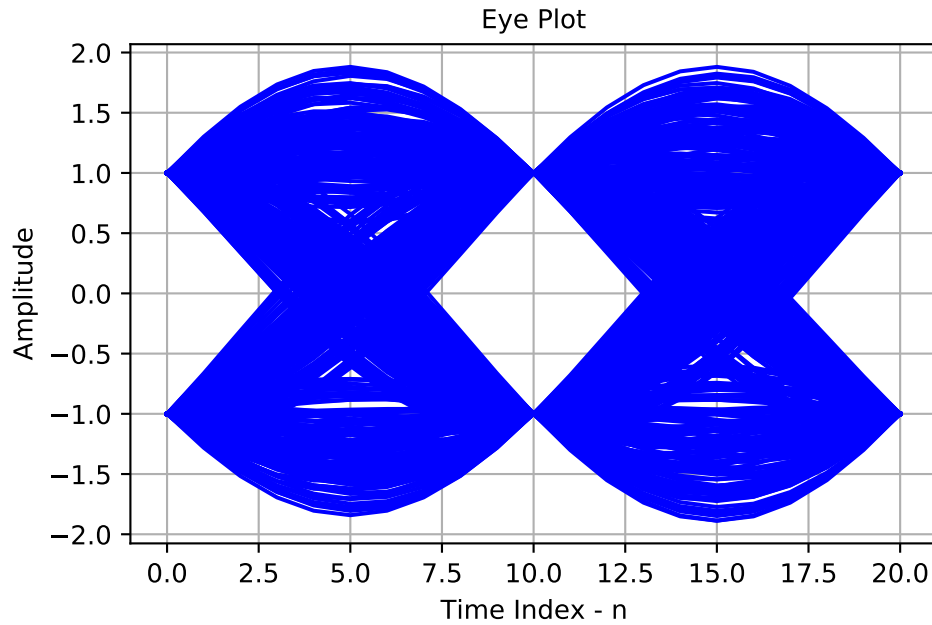
r [pole radius for IIR version, default = 0.95]

Returns

b [numerator coefficient ndarray]

a [denominator coefficient ndarray]

Notes

If the pole radius is 0 then an FIR version is created, that is there are no poles except at z = 0.

Examples

```
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm import sigsys as ss
```

```
>>> b_FIR, a_FIR = ss.fir_iir_notch(1000,8000,0)
>>> ss.zplane(b_FIR, a_FIR)
>>> plt.show()
```

```
>>> b_IIR, a_IIR = ss.fir_iir_notch(1000,8000)
>>> ss.zplane(b_IIR, a_IIR)
```

sk_dsp_comm.sigsys.**from_wav**(*filename*)

Read a wave file.

A wrapper function for scipy.io.wavfile.read that also includes int16 to float [-1,1] scaling.
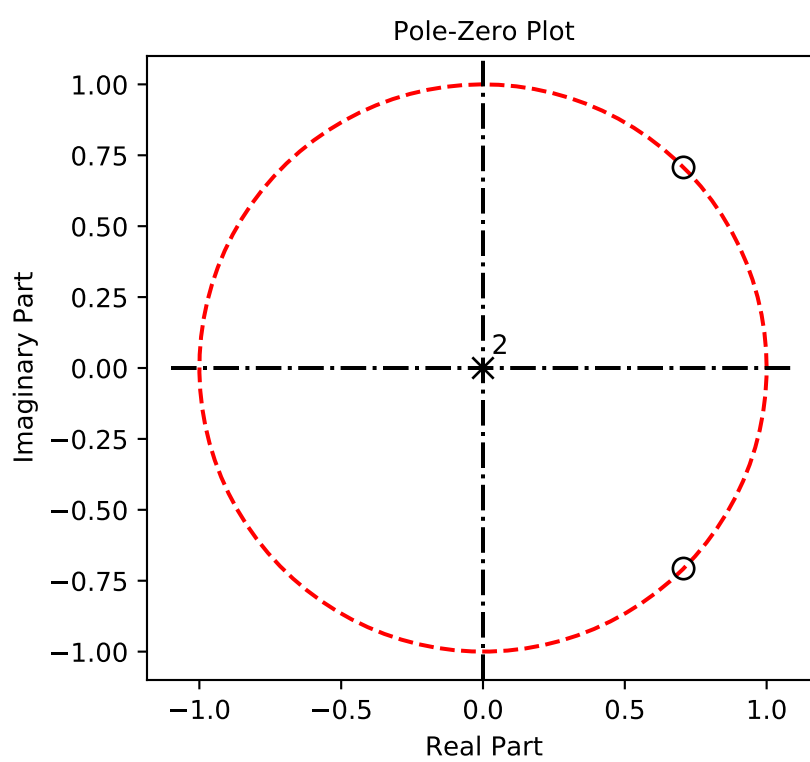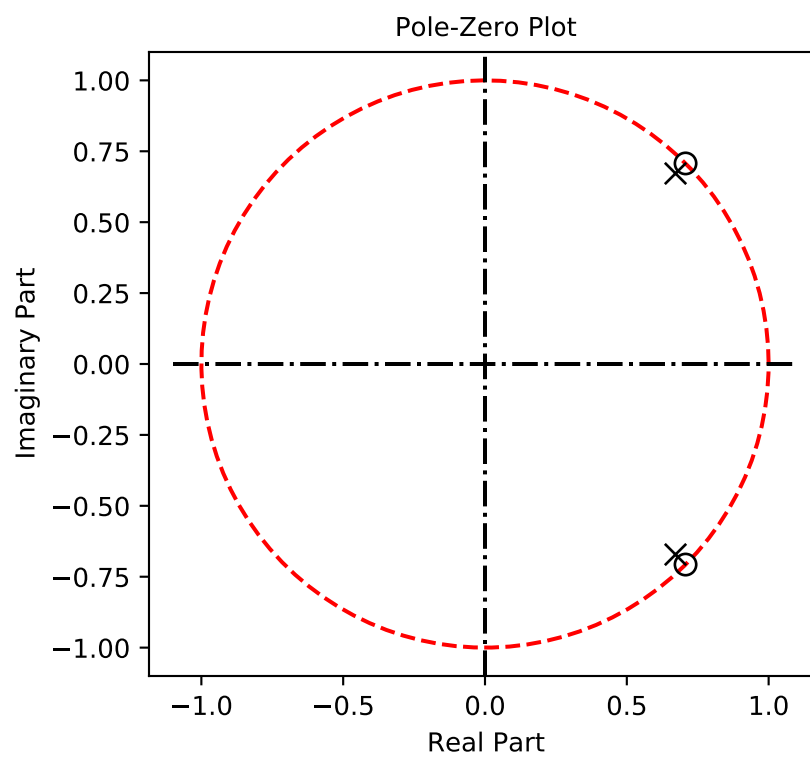
> **Parameters**

>> **filename** [file name string]

> **Returns**

>> **fs** [sampling frequency in Hz]

>> **x** [ndarray of normalized to 1 signal samples]

Examples

```
>>> fs,x = from_wav('test_file.wav')
```

sk_dsp_comm.sigsys.**fs_approx**(*Xk*, *fk*, *t*)

> Synthesize periodic signal x(t) using Fourier series coefficients at harmonic frequencies

> Assume the signal is real so coefficients Xk are supplied for nonnegative indicies. The negative index coefficients are assumed to be complex conjugates.

>> **Parameters**

>>> **Xk** [ndarray of complex Fourier series coefficients]

>>> **fk** [ndarray of harmonic frequencies in Hz]

>>> **t** [ndarray time axis corresponding to output signal array x_approx]

>> **Returns**

>>> **x_approx** [ndarray of periodic waveform approximation over time span t]

Examples

```
>>> t = arange(0,2,.002)
>>> # a 20% duty cycle pulse train
>>> n = arange(0,20,1) # 0 to 19th harmonic
>>> fk = 1*n % period = 1s
>>> t, x_approx = fs_approx(Xk,fk,t)
>>> plot(t,x_approx)
```

sk_dsp_comm.sigsys.**fs_coeff**(*xp*, *N*, *f0*, *one_side=True*)

> Numerically approximate the Fourier series coefficients given periodic x(t).

> The input is assumed to represent one period of the waveform x(t) that has been uniformly sampled. The number of samples supplied to represent one period of the waveform sets the sampling rate.

>> **Parameters**

>>> **xp** [ndarray of one period of the waveform x(t)]

>>> **N** [maximum Fourier series coefficient, [0,. . .,N]]

>>> **f0** [fundamental frequency used to form fk.]

>> **Returns**

>>> **Xk** [ndarray of the coefficients over indices [0,1,. . .,N]]

>>> **fk** [ndarray of the harmonic frequencies [0, f0,2f0,. . .,Nf0]]

Notes

len(xp) >= 2*N+1 as len(xp) is the fft length.

Examples

```
>>> import matplotlib.pyplot as plt
>>> from numpy import arange
>>> import sk_dsp_comm.sigsys as ss
>>> t = arange(0,1,1/1024.)
>>> # a 20% duty cycle pulse starting at t = 0
>>> x_rect = ss.rect(t-.1,0.2)
>>> Xk, fk = ss.fs_coeff(x_rect,25,10)
>>> # plot the spectral lines
>>> ss.line_spectra(fk,Xk,'mag')
>>> plt.show()
```
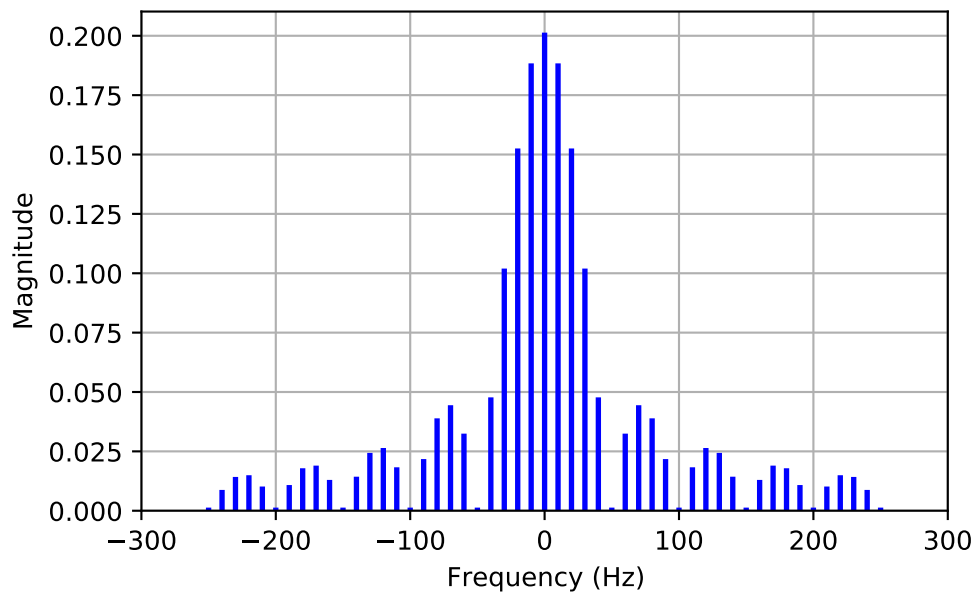

sk_dsp_comm.sigsys.**ft_approx**(*x*, *t*, *Nfft*)

Approximate the Fourier transform of a finite duration signal using scipy.signal.freqz()

> **Parameters**
>
>> **x** [input signal array]
>>
>> **t** [time array used to create x(t)]
>>
>> **Nfft** [the number of frdquency domain points used to] approximate X(f) on the interval [fs/2,fs/2], where fs = 1/Dt. Dt being the time spacing in array t
>
> **Returns**

f [frequency axis array in Hz]

X [the Fourier transform approximation (complex)]

Notes

The output time axis starts at the sum of the starting values in x1 and x2 and ends at the sum of the two ending values in x1 and x2. The default extents of ('f','f') are used for signals that are active (have support) on or within n1 and n2 respectively. A right-sided signal such as $a^n * u[n]$ is semi-infinite, so it has extent 'r' and the convolution output will be truncated to display only the valid results.

Examples

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> import sk_dsp_comm.sigsys as ss
>>> fs = 100 # sampling rate in Hz
>>> tau = 1
>>> t = np.arange(-5,5,1/fs)
>>> x0 = ss.rect(t-.5,tau)
>>> plt.figure(figsize=(6,5))
>>> plt.plot(t,x0)
>>> plt.grid()
>>> plt.ylim([-0.1,1.1])
>>> plt.xlim([-2,2])
>>> plt.title(r'Exact Waveform')
>>> plt.xlabel(r'Time (s)')
>>> plt.ylabel(r'$x_0(t)$')
>>> plt.show()
```

```
>>> # FT Exact Plot
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> import sk_dsp_comm.sigsys as ss
>>> fs = 100 # sampling rate in Hz
>>> tau = 1
>>> t = np.arange(-5,5,1/fs)
>>> x0 = ss.rect(t-.5,tau)
>>> fe = np.arange(-10,10,.01)
>>> X0e = tau*np.sinc(fe*tau)
>>> plt.plot(fe,abs(X0e))
>>> #plot(f,angle(X0))
>>> plt.grid()
>>> plt.xlim([-10,10])
>>> plt.title(r'Exact (Theory) Spectrum Magnitude')
>>> plt.xlabel(r'Frequency (Hz)')
>>> plt.ylabel(r'$|X_0e(f)|$')
>>> plt.show()
```

```
>>> # FT Approximation Plot
>>> import matplotlib.pyplot as plt
```
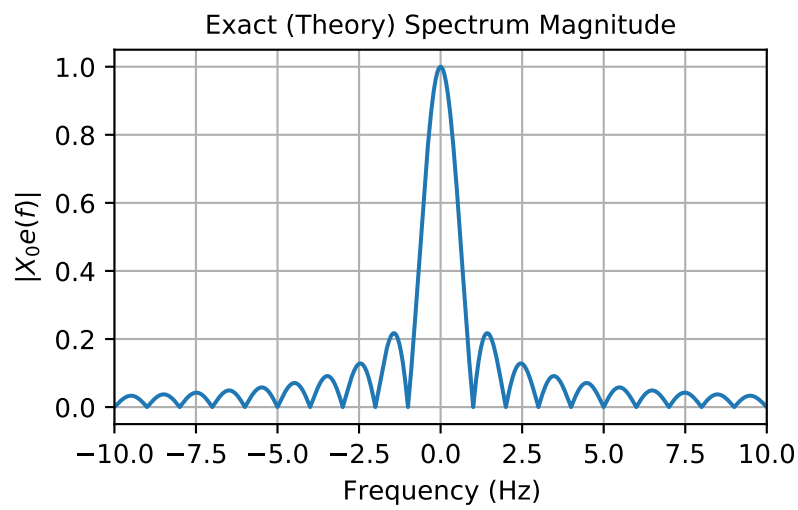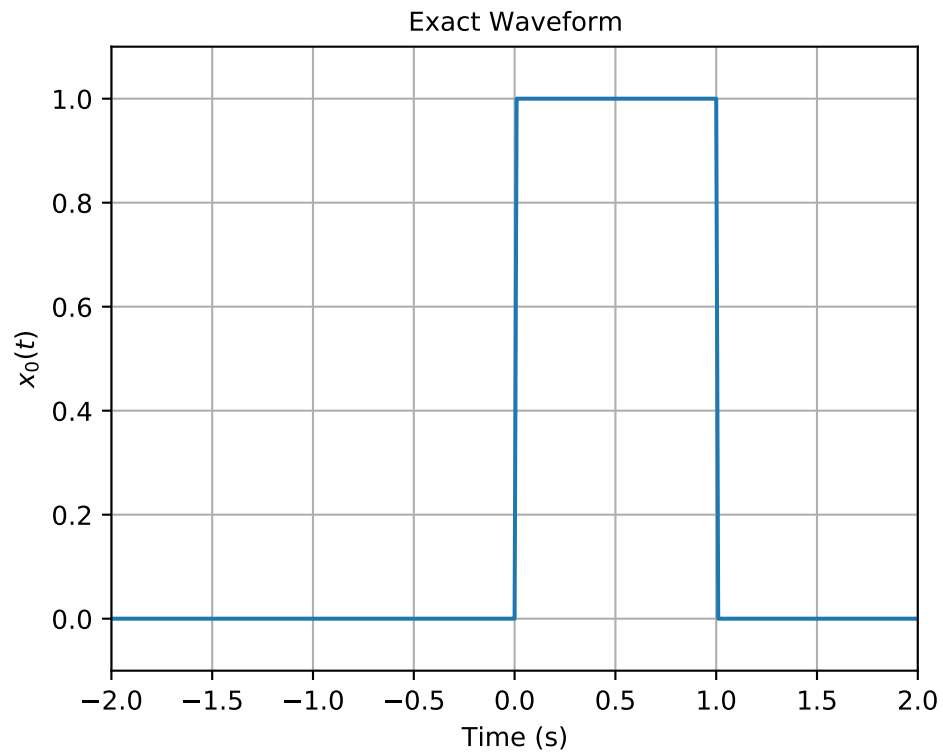
Exact Waveform

Exact (Theory) Spectrum Magnitude

```python
>>> import numpy as np
>>> import sk_dsp_comm.sigsys as ss
>>> fs = 100 # sampling rate in Hz
>>> tau = 1
>>> t = np.arange(-5,5,1/fs)
>>> x0 = ss.rect(t-.5,tau)
>>> f,X0 = ss.ft_approx(x0,t,4096)
>>> plt.plot(f,abs(X0))
>>> #plt.plot(f,angle(X0))
>>> plt.grid()
>>> plt.xlim([-10,10])
>>> plt.title(r'Approximation Spectrum Magnitude')
>>> plt.xlabel(r'Frequency (Hz)')
>>> plt.ylabel(r'$|X_0(f)|$');
>>> plt.tight_layout()
>>> plt.show()
```
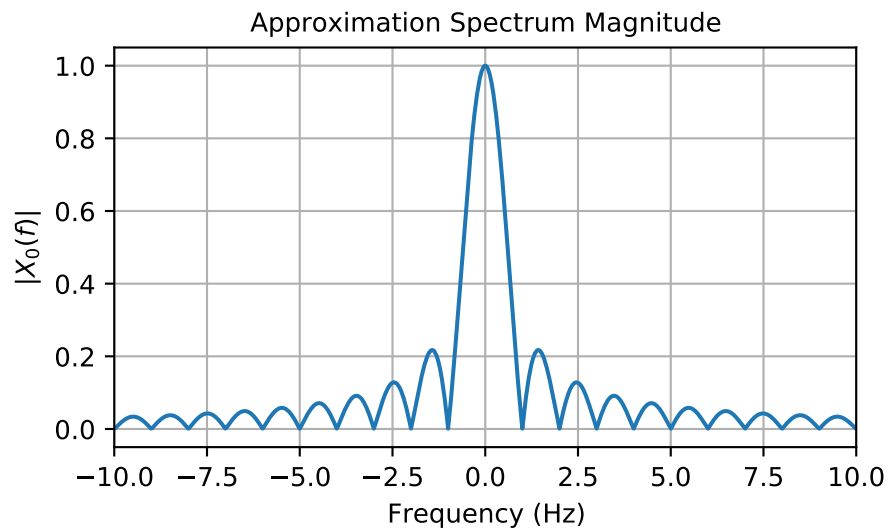

sk_dsp_comm.sigsys.**interp24**(x)

> Interpolate by L = 24 using Butterworth filters.

> The interpolation is done using three stages. Upsample by L = 2 and lowpass filter, upsample by 3 and lowpass filter, then upsample by L = 4 and lowpass filter. In all cases the lowpass filter is a 10th-order Butterworth lowpass.

> > **Parameters**

> > > **x** [ndarray of the input signal]

> > **Returns**

> > > **y** [ndarray of the output signal]

Notes

The cutoff frequency of the lowpass filters is 1/2, 1/3, and 1/4 to track the upsampling by 2, 3, and 4 respectively.

Examples

```
>>> y = interp24(x)
```

sk_dsp_comm.sigsys.**line_spectra**(*fk*, *Xk*, *mode*, *sides=2*, *linetype='b'*, *lwidth=2*, *floor_dB=- 100*, *fsize=(6, 4)*)

Plot the Fourier series line spectral given the coefficients.

This function plots two-sided and one-sided line spectra of a periodic signal given the complex exponential Fourier series coefficients and the corresponding harmonic frequencies.

> **Parameters**
>
>> **fk** [vector of real sinusoid frequencies]
>>
>> **Xk** [magnitude and phase at each positive frequency in fk]
>>
>> **mode** ['mag' => magnitude plot, 'magdB' => magnitude in dB plot,]
>>
>> **mode cont** ['magdBn' => magnitude in dB normalized, 'phase' => a phase plot in radians]
>>
>> **sides** [2; 2-sided or 1-sided]
>>
>> **linetype** [line type per Matplotlib definitions, e.g., 'b';]
>>
>> **lwidth** [2; linewidth in points]
>>
>> **fsize** [optional figure size in inches, default = (6,4) inches]
>
> **Returns**
>
>> **Nothing** [A plot window opens containing the line spectrum plot]

Notes

Since real signals are assumed the frequencies of fk are 0 and/or positive numbers. The supplied Fourier coefficients correspond.

Examples

```python
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from sk_dsp_comm.sigsys import line_spectra
>>> n = np.arange(0,25)
>>> # a pulse train with 10 Hz fundamental and 20% duty cycle
>>> fk = n*10
>>> Xk = np.sinc(n*10*.02)*np.exp(-1j*2*np.pi*n*10*.01) # 1j = sqrt(-1)
```

```python
>>> line_spectra(fk,Xk,'mag')
>>> plt.show()
```

```python
>>> line_spectra(fk,Xk,'phase')
```
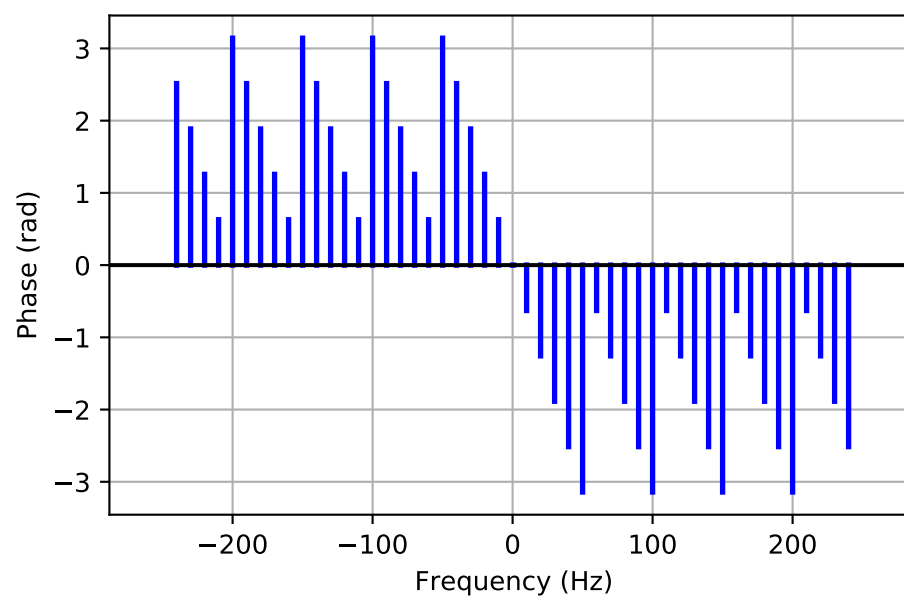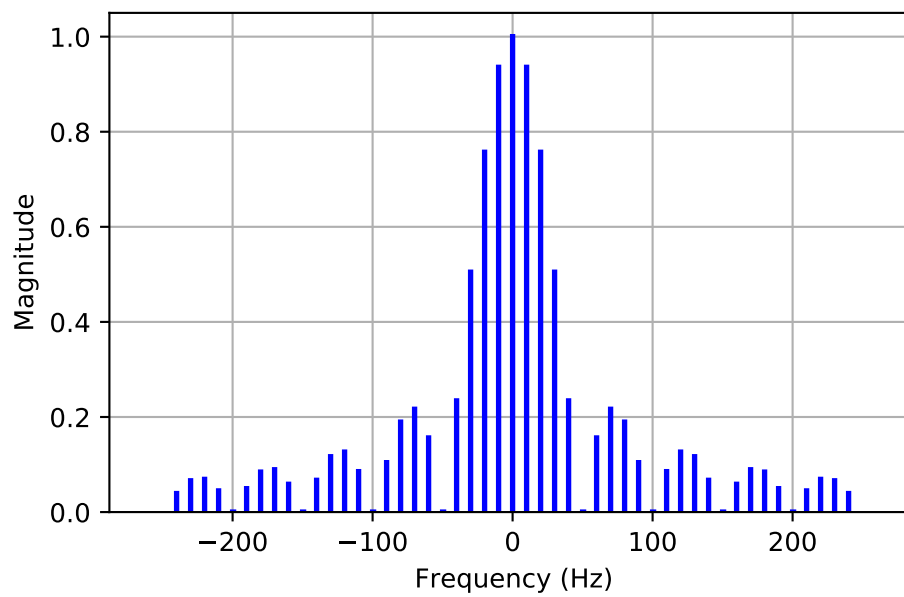
`sk_dsp_comm.sigsys.`**`lms_ic`**(*r*, *M*, *mu*, *delta=1*)
> Least mean square (LMS) interference canceller adaptive filter.

> A complete LMS adaptive filter simulation function for the case of interference cancellation. Used in the digital filtering case study.

> > **Parameters**

> > > **M** [FIR Filter length (order M-1)]

> > > **delta** [Delay used to generate the reference signal]

> > > **mu** [LMS step-size]

> > > **delta** [decorrelation delay between input and FIR filter input]

> > **Returns**

> > > **n** [ndarray Index vector]

> > > **r** [ndarray noisy (with interference) input signal]

> > > **r_hat** [ndarray filtered output (NB_hat[n])]

> > > **e** [ndarray error sequence (WB_hat[n])]

> > > **ao** [ndarray final value of weight vector]

> > > **F** [ndarray frequency response axis vector]

> > > **Ao** [ndarray frequency response of filter]

> **Examples**

```
>>> # import a speech signal
>>> fs,s = from_wav('OSR_us_000_0030_8k.wav')
>>> # add interference at 1kHz and 1.5 kHz and
>>> # truncate to 5 seconds
>>> r = soi_snoi_gen(s,10,5*8000,[1000, 1500])
>>> # simulate with a 64 tap FIR and mu = 0.005
>>> n,r,r_hat,e,ao,F,Ao = lms_ic(r,64,0.005)
```

`sk_dsp_comm.sigsys.`**`lp_samp`**(*fb*, *fs*, *fmax*, *N*, *shape='tri'*, *fsize=(6, 4)*)
> Lowpass sampling theorem plotting function.

> Display the spectrum of a sampled signal after setting the bandwidth, sampling frequency, maximum display frequency, and spectral shape.

> > **Parameters**

> > > **fb** [spectrum lowpass bandwidth in Hz]

> > > **fs** [sampling frequency in Hz]

> > > **fmax** [plot over [-fmax,fmax]]

> > > **shape** ['tri' or 'line']

> > > **N** [number of translates, N positive and N negative]

> > > **fsize** [the size of the figure window, default (6,4)]

> > **Returns**

> > > **Nothing** [A plot window opens containing the spectrum plot]

Examples

```
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm.sigsys import lp_samp
```

No aliasing as bandwidth 10 Hz < 25/2; fs > fb.

```
>>> lp_samp(10,25,50,10)
>>> plt.show()
```
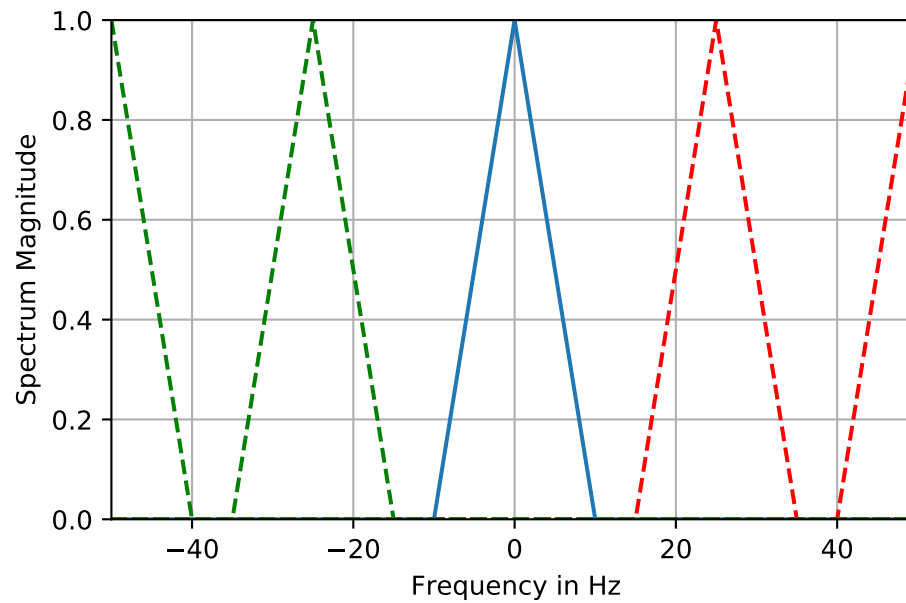

Now aliasing as bandwidth 15 Hz > 25/2; fs < fb.

```
>>> lp_samp(15,25,50,10)
```

sk_dsp_comm.sigsys.**lp_tri**(*f*, *fb*)
    Triangle spectral shape function used by *lp_samp()*.

    Parameters

        f [ndarray containing frequency samples]

        fb [the bandwidth as a float constant]

    Returns

        x [ndarray of spectrum samples for a single triangle shape]
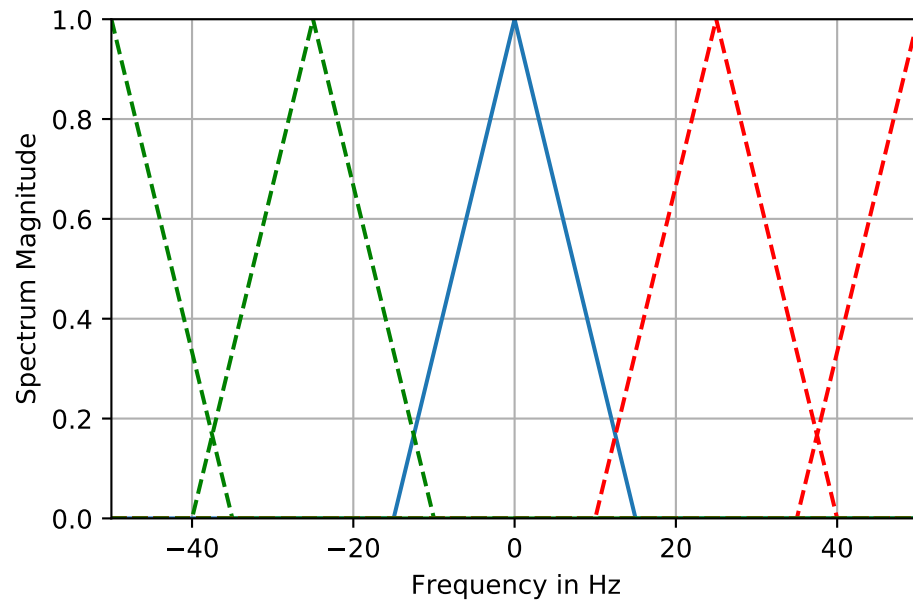
Notes

This is a support function for the lowpass spectrum plotting function `lp_samp()`.

Examples

```
>>> x = lp_tri(f, fb)
```

sk_dsp_comm.sigsys.**m_seq**(*m*)

Generate an m-sequence ndarray using an all-ones initialization.

Available m-sequence (PN generators) include m = 2,3,…,12, & 16.

> **Parameters**
>
> > **m** [the number of shift registers. 2,3, .., 12, & 16]
>
> **Returns**
>
> > **c** [ndarray of one period of the m-sequence]

Notes

The sequence period is 2**m - 1 (2^m - 1).

Examples

```
>>> c = m_seq(5)
```

`sk_dsp_comm.sigsys.`**`my_psd`**(*x*, *NFFT=1024*, *Fs=1*)

A local version of NumPy's PSD function that returns the plot arrays.

A mlab.psd wrapper function that returns two ndarrays; makes no attempt to auto plot anything.

> **Parameters**
>
> > **x** [ndarray input signal]
> >
> > **NFFT** [a power of two, e.g., 2**10 = 1024]
> >
> > **Fs** [the sampling rate in Hz]
>
> **Returns**
>
> > **Px** [ndarray of the power spectrum estimate]
> >
> > **f** [ndarray of frequency values]

Notes

This function makes it easier to overlay spectrum plots because you have better control over the axis scaling than when using psd() in the autoscale mode.

Examples

```
>>> import matplotlib.pyplot as plt
>>> from numpy import log10
>>> from sk_dsp_comm import sigsys as ss
>>> x,b, data = ss.nrz_bits(10000,10)
>>> Px,f = ss.my_psd(x,2**10,10)
>>> plt.plot(f, 10*log10(Px))
>>> plt.ylabel("Power Spectral Density (dB)")
>>> plt.xlabel("Frequency (Hz)")
>>> plt.show()
```

`sk_dsp_comm.sigsys.`**`nrz_bits`**(*n_bits*, *ns*, *pulse='rect'*, *alpha=0.25*, *m=6*)

Generate non-return-to-zero (NRZ) data bits with pulse shaping.

A baseband digital data signal using +/-1 amplitude signal values and including pulse shaping.

> **Parameters**
>
> > **n_bits** [number of NRZ +/-1 data bits to produce]
> >
> > **ns** [the number of samples per bit,]
> >
> > **pulse_type** ['rect' , 'rc', 'src' (default 'rect')]
> >
> > **alpha** [excess bandwidth factor(default 0.25)]

> **m** [single sided pulse duration (default = 6)]
>
> **Returns**
>
>> **x** [ndarray of the NRZ signal values]
>>
>> **b** [ndarray of the pulse shape]
>>
>> **data** [ndarray of the underlying data bits]

Notes

Pulse shapes include 'rect' (rectangular), 'rc' (raised cosine), 'src' (root raised cosine). The actual pulse length is 2*M+1 samples. This function is used by BPSK_tx in the Case Study article.

Examples

```
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm.sigsys import nrz_bits
>>> from numpy import arange
>>> x,b,data = nrz_bits(100, 10)
>>> t = arange(len(x))
>>> plt.plot(t, x)
>>> plt.ylim([-1.01, 1.01])
>>> plt.show()
```

sk_dsp_comm.sigsys.**nrz_bits2**(*data*, *Ns*, *pulse='rect'*, *alpha=0.25*, *M=6*)

Generate non-return-to-zero (NRZ) data bits with pulse shaping with user data

A baseband digital data signal using +/-1 amplitude signal values and including pulse shaping. The data sequence is user supplied.

> **Parameters**
>
>> **data** [ndarray of the data bits as 0/1 values]
>>
>> **Ns** [the number of samples per bit,]
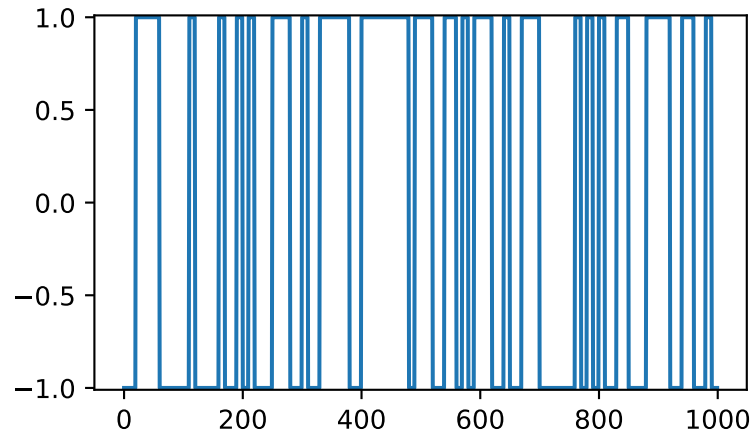
> **pulse_type** ['rect' , 'rc', 'src' (default 'rect')]
>
> **alpha** [excess bandwidth factor(default 0.25)]
>
> **M** [single sided pulse duration (default = 6)]

Returns

> **x** [ndarray of the NRZ signal values]
>
> **b** [ndarray of the pulse shape]

Notes

Pulse shapes include 'rect' (rectangular), 'rc' (raised cosine), 'src' (root raised cosine). The actual pulse length is 2*M+1 samples.

Examples

```
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm.sigsys import nrz_bits2
>>> from sk_dsp_comm.sigsys import m_seq
>>> from numpy import arange
>>> x,b = nrz_bits2(m_seq(5),10)
>>> t = arange(len(x))
>>> plt.ylim([-1.01, 1.01])
>>> plt.plot(t,x)
```

sk_dsp_comm.sigsys.**oa_filter**(*x*, *h*, *N*, *mode=0*)

> Overlap and add transform domain FIR filtering.
>
> This function implements the classical overlap and add method of transform domain filtering using a length P FIR filter.
>
> > **Parameters**
> >
> > > **x** [input signal to be filtered as an ndarray]

h [FIR filter coefficients as an ndarray of length P]

N [FFT size > P, typically a power of two]

mode [0 or 1, when 1 returns a diagnostic matrix]

Returns

y [the filtered output as an ndarray]

y_mat [an ndarray whose rows are the individual overlap outputs.]

Notes
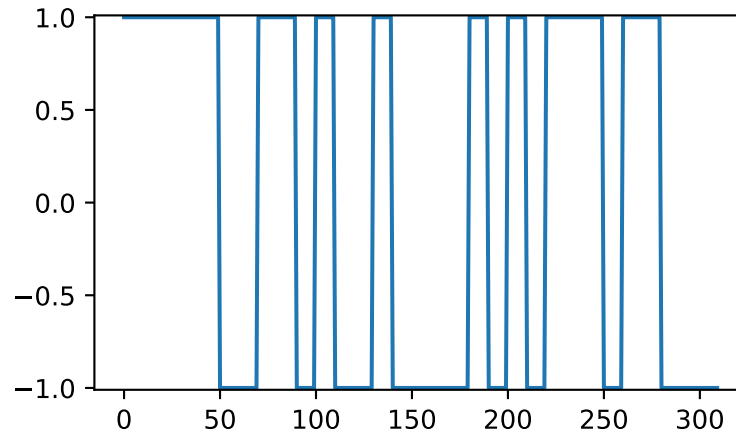
y_mat is used for diagnostics and to gain understanding of the algorithm.

Examples

```
>>> import numpy as np
>>> from sk_dsp_comm.sigsys import oa_filter
>>> n = np.arange(0,100)
>>> x = np.cos(2*np.pi*0.05*n)
>>> b = np.ones(10)
>>> y = oa_filter(x,h,N)
>>> # set mode = 1
>>> y, y_mat = oa_filter(x,h,N,1)
```

sk_dsp_comm.sigsys.**os_filter**(*x, h, N, mode=0*)

Overlap and save transform domain FIR filtering.

This function implements the classical overlap and save method of transform domain filtering using a length P FIR filter.

Parameters

x [input signal to be filtered as an ndarray]

h [FIR filter coefficients as an ndarray of length P]

N [FFT size > P, typically a power of two]

mode [0 or 1, when 1 returns a diagnostic matrix]

Returns

y [the filtered output as an ndarray]

y_mat [an ndarray whose rows are the individual overlap outputs.]

Notes

y_mat is used for diagnostics and to gain understanding of the algorithm.

Examples

```
>>> from numpy import arange, cos, pi, ones
>>> n = arange(0,100)
>>> x = cos(2*pi*0.05*n)
>>> b = ones(10)
>>> y = os_filter(x,h,N)
>>> # set mode = 1
>>> y, y_mat = os_filter(x,h,N,1)
```

sk_dsp_comm.sigsys.**peaking**(*GdB, fc, Q=3.5, fs=44100.0*)

A second-order peaking filter having GdB gain at fc and approximately and 0 dB otherwise.

The filter coefficients returns correspond to a biquadratic system function containing five parameters.

Parameters

GdB [Lowpass gain in dB]

fc [Center frequency in Hz]

Q [Filter Q which is inversely proportional to bandwidth]

fs [Sampling frquency in Hz]

Returns

b [ndarray containing the numerator filter coefficients]

a [ndarray containing the denominator filter coefficients]

Examples

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from sk_dsp_comm.sigsys import peaking
>>> from scipy import signal
>>> b,a = peaking(2.0,500)
>>> f = np.logspace(1,5,400)
>>> w,H = signal.freqz(b,a,2*np.pi*f/44100)
>>> plt.semilogx(f,20*np.log10(abs(H)))
```

```
>>> plt.ylabel("Power Spectral Density (dB)")
>>> plt.xlabel("Frequency (Hz)")
>>> plt.show()
```
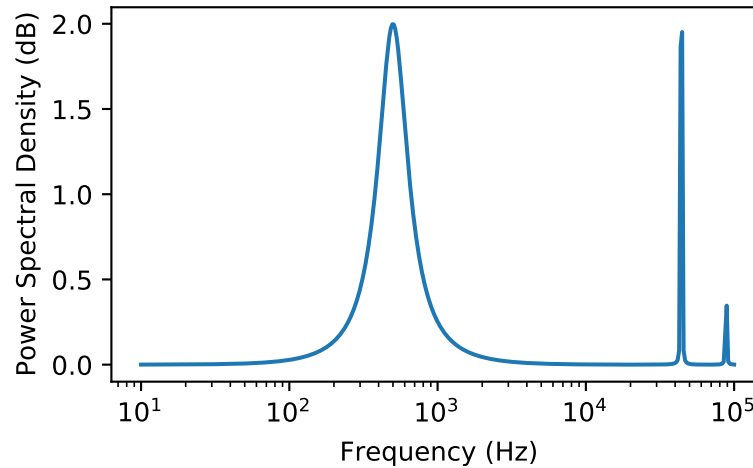


```
>>> b,a = peaking(-5.0,500,4)
>>> w,H = signal.freqz(b,a,2*np.pi*f/44100)
>>> plt.semilogx(f,20*np.log10(abs(H)))
>>> plt.ylabel("Power Spectral Density (dB)")
>>> plt.xlabel("Frequency (Hz)")
```
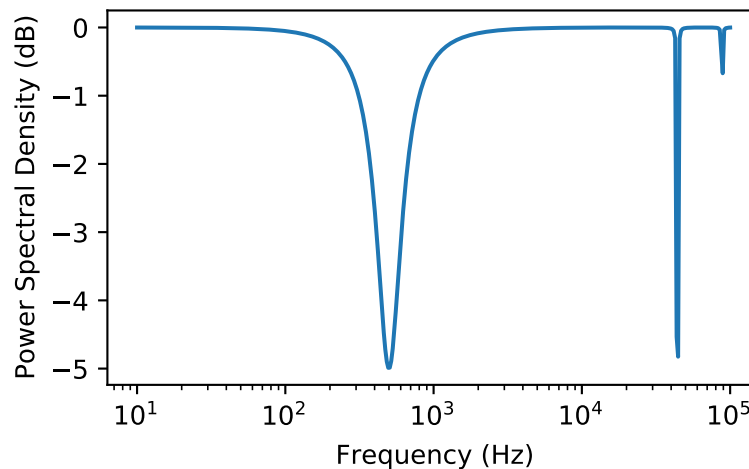

sk_dsp_comm.sigsys.**pn_gen**(*n_bits*, *m=5*)

Maximal length sequence signal generator.

Generates a sequence 0/1 bits of N_bit duration. The bits themselves are obtained from an m-sequence of length

m. Available m-sequence (PN generators) include m = 2,3,. . .,12, & 16.

> **Parameters**
>
> > **n_bits** [the number of bits to generate]
> >
> > **m** [the number of shift registers. 2,3, .., 12, & 16]
>
> **Returns**
>
> > **PN** [ndarray of the generator output over N_bits]

Notes

The sequence is periodic having period 2**m - 1 (2^m - 1).

Examples

```
>>> # A 15 bit period signal nover 50 bits
>>> PN = pn_gen(50,4)
```

sk_dsp_comm.sigsys.**position_cd**(*Ka*, *out_type='fb_exact'*)

> CD sled position control case study of Chapter 18.
>
> The function returns the closed-loop and open-loop system function for a CD/DVD sled position control system. The loop amplifier gain is the only variable that may be changed. The returned system function can however be changed.
>
> > **Parameters**
> >
> > > **Ka** [loop amplifier gain, start with 50.]
> > >
> > > **out_type** ['open_loop' for open loop system function]
> > >
> > > **out_type** ['fb_approx' for closed-loop approximation]
> > >
> > > **out_type** ['fb_exact' for closed-loop exact]
> >
> > **Returns**
> >
> > > **b** [numerator coefficient ndarray]
> > >
> > > **a** [denominator coefficient ndarray]

Notes

With the exception of the loop amplifier gain, all other parameters are hard-coded from Case Study example.

Examples

```
>>> b,a = position_cd(Ka,'fb_approx')
>>> b,a = position_cd(Ka,'fb_exact')
```

sk_dsp_comm.sigsys.**prin_alias**(*f_in*, *fs*)

> Calculate the principle alias frequencies.
>
> Given an array of input frequencies the function returns an array of principle alias frequencies.
>
> > **Parameters**

>
> **f_in** [ndarray of input frequencies]
>
> **fs** [sampling frequency]
>
> **Returns**
>
> > **f_out** [ndarray of principle alias frequencies]

Examples

```
>>> # Linear frequency sweep from 0 to 50 Hz
>>> f_in = arange(0,50,0.1)
>>> # Calculate principle alias with fs = 10 Hz
>>> f_out = prin_alias(f_in,10)
```

sk_dsp_comm.sigsys.**rc_imp**(*Ns*, *alpha*, *M=6*)

A truncated raised cosine pulse used in digital communications.

The pulse shaping factor $0 < \alpha < 1$ is required as well as the truncation factor M which sets the pulse duration to be 2*M*Tsymbol.

> **Parameters**
>
> > **Ns** [number of samples per symbol]
> >
> > **alpha** [excess bandwidth factor on (0, 1), e.g., 0.35]
> >
> > **M** [equals RC one-sided symbol truncation factor]
>
> **Returns**
>
> > **b** [ndarray containing the pulse shape]

Notes

The pulse shape b is typically used as the FIR filter coefficients when forming a pulse shaped digital communications waveform.

Examples

Ten samples per symbol and alpha = 0.35.

```
>>> import matplotlib.pyplot as plt
>>> from numpy import arange
>>> from sk_dsp_comm.sigsys import rc_imp
>>> b = rc_imp(10,0.35)
>>> n = arange(-10*6,10*6+1)
>>> plt.stem(n,b)
>>> plt.show()
```

sk_dsp_comm.sigsys.**rect**(*t*, *tau*)

Approximation to the rectangle pulse Pi(t/tau).

In this numerical version of Pi(t/tau) the pulse is active over -tau/2 <= t <= tau/2.

> **Parameters**
>
> > **t** [ndarray of the time axis]
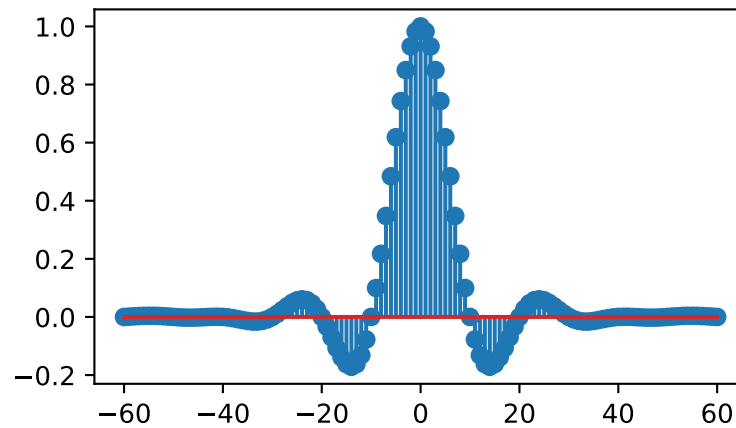
tau [the pulse width]

Returns

x [ndarray of the signal Pi(t/tau)]

Examples

```
>>> import matplotlib.pyplot as plt
>>> from numpy import arange
>>> from sk_dsp_comm.sigsys import rect
>>> t = arange(-1,5,.01)
>>> x = rect(t,1.0)
>>> plt.plot(t,x)
>>> plt.ylim([0, 1.01])
>>> plt.show()
```

To turn on the pulse at t = 1 shift t.

```
>>> x = rect(t - 1.0,1.0)
>>> plt.plot(t,x)
>>> plt.ylim([0, 1.01])
```

sk_dsp_comm.sigsys.**rect_conv**(*n, n_len*)

The theoretical result of convolving two rectangle sequences.

The result is a triangle. The solution is based on pure analysis. Simply coded as opposed to efficiently coded.

Parameters

**n** [ndarray of time axis]

n_len [rectangle pulse duration]

Returns

y [ndarray of of output signal]
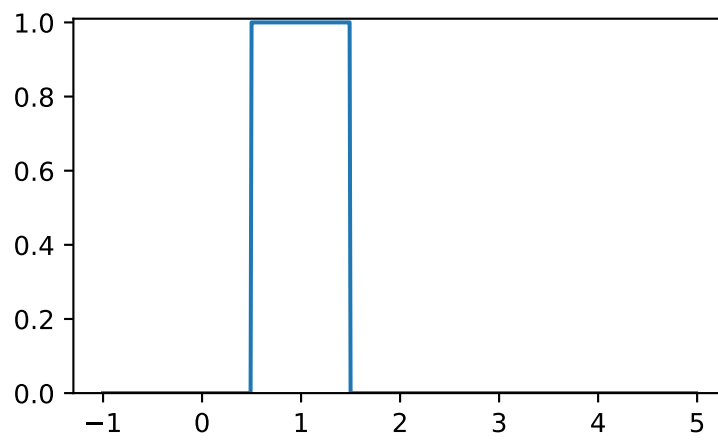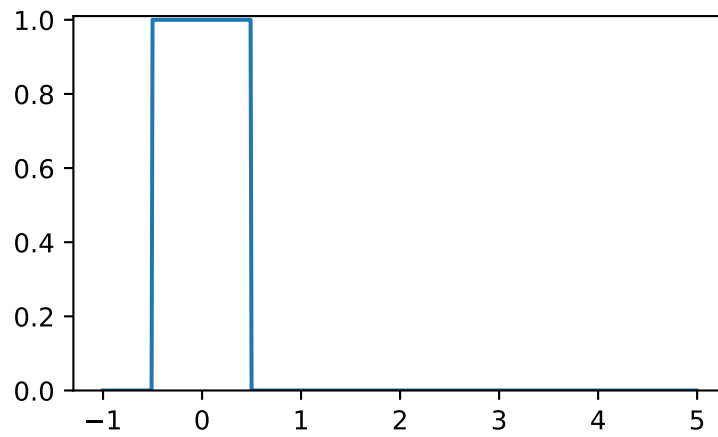
Examples

```
>>> import matplotlib.pyplot as plt
>>> from numpy import arange
>>> from sk_dsp_comm.sigsys import rect_conv
>>> n = arange(-5,20)
>>> y = rect_conv(n,6)
>>> plt.plot(n, y)
>>> plt.show()
```
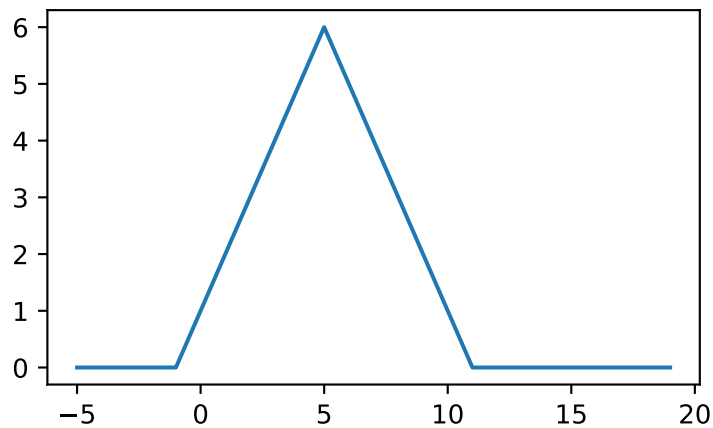

sk_dsp_comm.sigsys.**scatter**(*x*, *ns*, *start*)

Sample a baseband digital communications waveform at the symbol spacing.

> **Parameters**
>
> > **x** [ndarray of the input digital comm signal]
> >
> > **ns** [number of samples per symbol (bit)]
> >
> > **start** [the array index to start the sampling]
>
> **Returns**
>
> > **xI** [ndarray of the real part of x following sampling]
> >
> > **xQ** [ndarray of the imaginary part of x following sampling]

Notes

Normally the signal is complex, so the scatter plot contains clusters at points in the complex plane. For a binary signal such as BPSK, the point centers are nominally +/-1 on the real axis. Start is used to eliminate transients from the FIR pulse shaping filters from appearing in the scatter plot.

Examples

```
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm import sigsys as ss
>>> x,b, data = ss.nrz_bits(1000,10,'rc')
>>> # Add some noise so points are now scattered about +/-1
>>> y = ss.cpx_awgn(x,20,10)
>>> yI,yQ = ss.scatter(y,10,60)
>>> plt.plot(yI,yQ,'.')
>>> plt.axis('equal')
>>> plt.ylabel("Quadrature")
>>> plt.xlabel("In-Phase")
>>> plt.grid()
>>> plt.show()
```
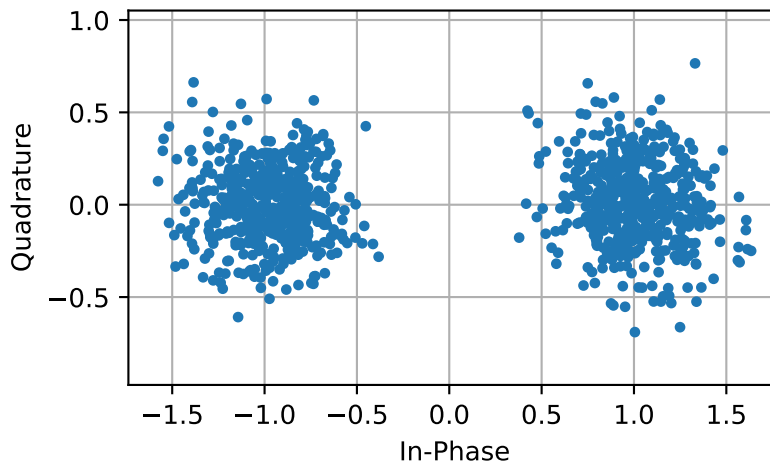

sk_dsp_comm.sigsys.**simple_quant**(*x*, *b_tot*, *x_max*, *limit*)

A simple rounding quantizer for bipolar signals having Btot = B + 1 bits.

This function models a quantizer that employs Btot bits that has one of three selectable limiting types: saturation, overflow, and none. The quantizer is bipolar and implements rounding.

> **Parameters**
>
> > **x** [input signal ndarray to be quantized]
> >
> > **b_tot** [total number of bits in the quantizer, e.g. 16]
> >
> > **x_max** [quantizer full-scale dynamic range is [-Xmax, Xmax]]
> >
> > **Limit = Limiting of the form 'sat', 'over', 'none'**
>
> **Returns**
>
> > **xq** [quantized output ndarray]

Notes

The quantization can be formed as e = xq - x

Examples

```
>>> import matplotlib.pyplot as plt
>>> from matplotlib.mlab import psd
>>> import numpy as np
>>> from sk_dsp_comm import sigsys as ss
>>> n = np.arange(0,10000)
>>> x = np.cos(2*np.pi*0.211*n)
>>> y = ss.sinusoid_awgn(x,90)
>>> Px, f = psd(y,2**10,Fs=1)
>>> plt.plot(f, 10*np.log10(Px))
>>> plt.ylim([-80, 25])
>>> plt.ylabel("Power Spectral Density (dB)")
>>> plt.xlabel(r'Normalized Frequency $\omega/2\pi$')
>>> plt.show()
```
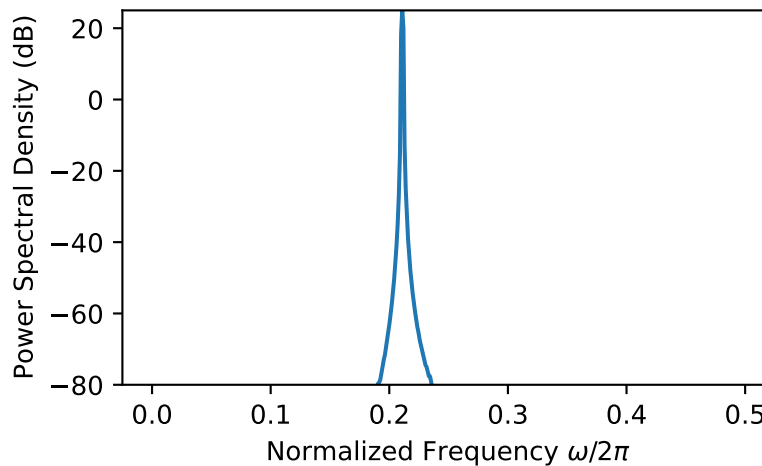


```
>>> yq = ss.simple_quant(y,12,1,'sat')
>>> Px, f = psd(yq,2**10,Fs=1)
>>> plt.plot(f, 10*np.log10(Px))
>>> plt.ylim([-80, 25])
>>> plt.ylabel("Power Spectral Density (dB)")
>>> plt.xlabel(r'Normalized Frequency $\omega/2\pi$')
>>> plt.show()
```

sk_dsp_comm.sigsys.**simple_sa**(*x, NS, NFFT, fs, NAVG=1, window='boxcar'*)
    Spectral estimation using windowing and averaging.

    This function implements averaged periodogram spectral estimation estimation similar to the NumPy's psd() function, but more specialized for the windowing case study of Chapter 16.
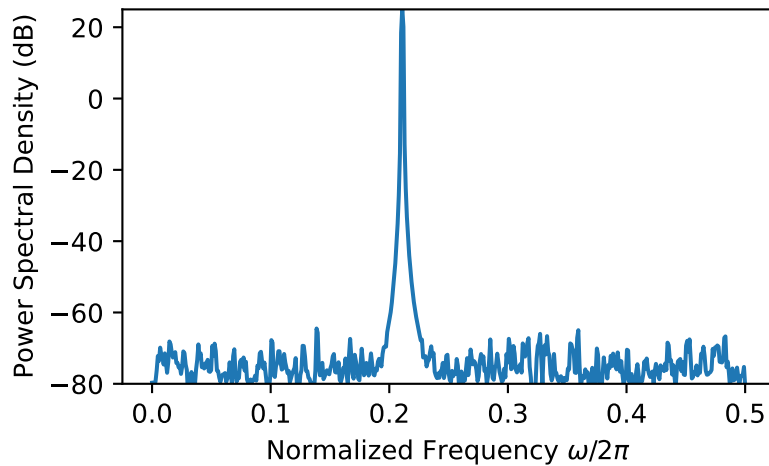
Parameters

> **x** [ndarray containing the input signal]
>
> **NS** [The subrecord length less zero padding, e.g. NS < NFFT]
>
> **NFFT** [FFT length, e.g., 1024 = 2**10]
>
> **fs** [sampling rate in Hz]
>
> **NAVG** [the number of averages, e.g., 1 for deterministic signals]
>
> **window** [hardcoded window 'boxcar' (default) or 'hanning']

Returns

> **f** [ndarray frequency axis in Hz on [0, fs/2]]
>
> **Sx** [ndarray the power spectrum estimate]

Notes

The function also prints the maximum number of averages K possible for the input data record.

Examples

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from sk_dsp_comm import sigsys as ss
>>> n = np.arange(0,2048)
>>> x = np.cos(2*np.pi*1000/10000*n) + 0.01*np.cos(2*np.pi*3000/10000*n)
>>> f, Sx = ss.simple_sa(x,128,512,10000)
>>> plt.plot(f, 10*np.log10(Sx))
>>> plt.ylim([-80, 0])
>>> plt.xlabel("Frequency (Hz)")
>>> plt.ylabel("Power Spectral Density (dB)")
>>> plt.show()
```
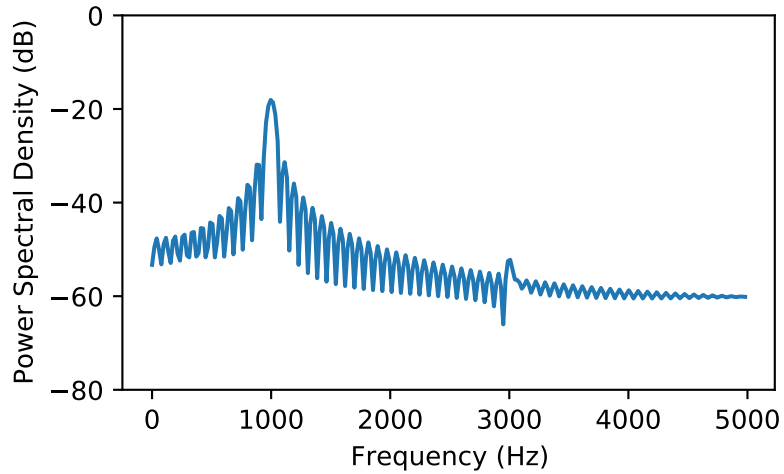
With a hanning window.

```
>>> f, Sx = ss.simple_sa(x,256,1024,10000,window='hanning')
>>> plt.plot(f, 10*np.log10(Sx))
>>> plt.xlabel("Frequency (Hz)")
>>> plt.ylabel("Power Spectral Density (dB)")
>>> plt.ylim([-80, 0])
```
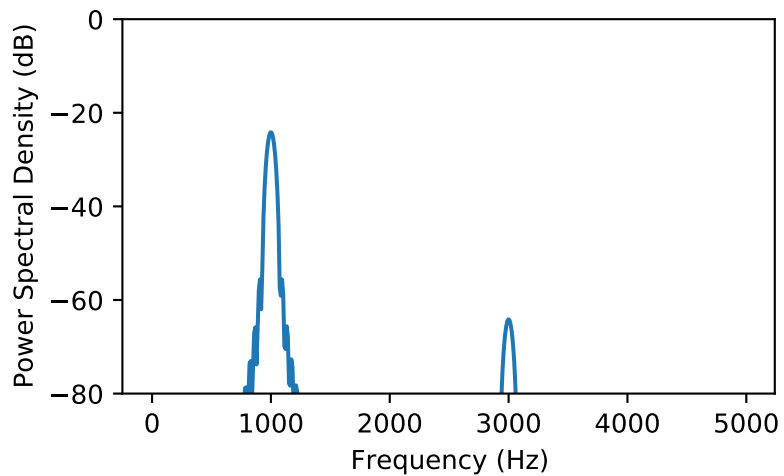

sk_dsp_comm.sigsys.**sinusoid_awgn**(*x*, *SNRdB*)

> Add white Gaussian noise to a single real sinusoid.
>
> Input a single sinusoid to this function and it returns a noisy sinusoid at a specific SNR value in dB. Sinusoid power is calculated using np.var.
>
> > **Parameters**
> >
> > > **x** [Input signal as ndarray consisting of a single sinusoid]

SNRdB [SNR in dB for output sinusoid]

Returns

y [Noisy sinusoid return vector]

Examples

```
>>> # set the SNR to 10 dB
>>> n = arange(0,10000)
>>> x = cos(2*pi*0.04*n)
>>> y = sinusoid_awgn(x,10.0)
```

sk_dsp_comm.sigsys.**soi_snoi_gen**(*s*, *SIR_dB*, *N*, *fi*, *fs=8000*)

Add an interfering sinusoidal tone to the input signal at a given SIR_dB.

The input is the signal of interest (SOI) and number of sinsuoid signals not of interest (SNOI) are addedto the SOI at a prescribed signal-to- intereference SIR level in dB.

Parameters

s [ndarray of signal of SOI]

SIR_dB [interference level in dB]

N [Trim input signal s to length N + 1 samples]

fi [ndarray of intereference frequencies in Hz]

fs [sampling rate in Hz, default is 8000 Hz]

Returns

r [ndarray of combined signal plus intereference of length N+1 samples]

Examples

```
>>> # load a speech ndarray and trim to 5*8000 + 1 samples
>>> fs,s = from_wav('OSR_us_000_0030_8k.wav')
>>> r = soi_snoi_gen(s,10,5*8000,[1000, 1500])
```

sk_dsp_comm.sigsys.**splane**(*b*, *a*, *auto_scale=True*, *size=[- 1, 1, - 1, 1]*)

Create an s-plane pole-zero plot.

As input the function uses the numerator and denominator s-domain system function coefficient ndarrays b and a respectively. Assumed to be stored in descending powers of s.

Parameters

b [numerator coefficient ndarray.]

a [denominator coefficient ndarray.]

auto_scale [True]

size [[xmin,xmax,ymin,ymax] plot scaling when scale = False]

Returns

(M,N) [tuple of zero and pole counts + plot window]

Notes

This function tries to identify repeated poles and zeros and will place the multiplicity number above and to the right of the pole or zero. The difficulty is setting the tolerance for this detection. Currently it is set at 1e-3 via the function signal.unique_roots.

Examples

```
>>> # Here the plot is generated using auto_scale
>>> splane(b,a)
>>> # Here the plot is generated using manual scaling
>>> splane(b,a,False,[-10,1,-10,10])
```

sk_dsp_comm.sigsys.**sqrt_rc_imp**(*Ns*, *alpha*, *M=6*)

A truncated square root raised cosine pulse used in digital communications.

The pulse shaping factor 0< alpha < 1 is required as well as the truncation factor M which sets the pulse duration to be 2*M*Tsymbol.

> **Parameters**
>
>> **Ns** [number of samples per symbol]
>>
>> **alpha** [excess bandwidth factor on (0, 1), e.g., 0.35]
>>
>> **M** [equals RC one-sided symbol truncation factor]
>
> **Returns**
>
>> **b** [ndarray containing the pulse shape]

Notes

The pulse shape b is typically used as the FIR filter coefficients when forming a pulse shaped digital communications waveform. When square root raised cosine (SRC) pulse is used generate Tx signals and at the receiver used as a matched filter (receiver FIR filter), the received signal is now raised cosine shaped, this having zero intersymbol interference and the optimum removal of additive white noise if present at the receiver input.

Examples

```
>>> # ten samples per symbol and alpha = 0.35
>>> import matplotlib.pyplot as plt
>>> from numpy import arange
>>> from sk_dsp_comm.sigsys import sqrt_rc_imp
>>> b = sqrt_rc_imp(10,0.35)
>>> n = arange(-10*6,10*6+1)
>>> plt.stem(n,b)
>>> plt.show()
```

sk_dsp_comm.sigsys.**step**(*t*)

Approximation to step function signal u(t).

In this numerical version of u(t) the step turns on at t = 0.

> **Parameters**
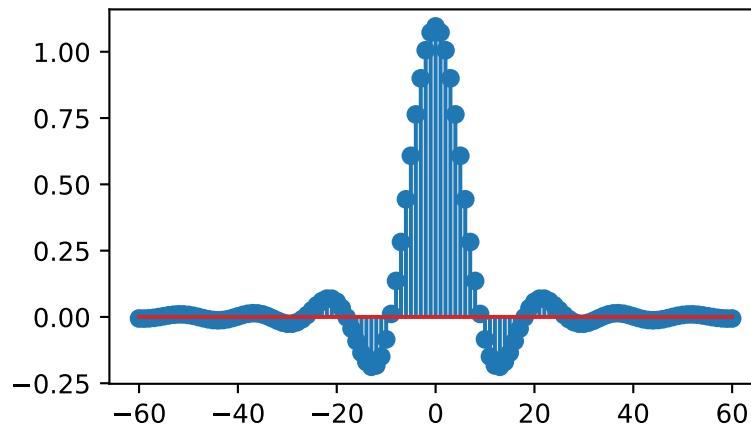
> **t** [ndarray of the time axis]
>
> **Returns**
>
> > **x** [ndarray of the step function signal u(t)]

Examples

```
>>> import matplotlib.pyplot as plt
>>> from numpy import arange
>>> from sk_dsp_comm.sigsys import step
>>> t = arange(-1,5,.01)
>>> x = step(t)
>>> plt.plot(t,x)
>>> plt.ylim([-0.01, 1.01])
>>> plt.show()
```

To turn on at t = 1, shift t.

```
>>> x = step(t - 1.0)
>>> plt.ylim([-0.01, 1.01])
>>> plt.plot(t,x)
```

sk_dsp_comm.sigsys.**ten_band_eq_filt**(*x*, *GdB*, *Q=3.5*)

> Filter the input signal x with a ten-band equalizer having octave gain values in ndarray GdB.
>
> The signal x is filtered using octave-spaced peaking filters starting at 31.25 Hz and stopping at 16 kHz. The Q of each filter is 3.5, but can be changed. The sampling rate is assumed to be 44.1 kHz.
>
> > **Parameters**
> >
> > > **x** [ndarray of the input signal samples]
> > >
> > > **GdB** [ndarray containing ten octave band gain values [G0dB,…,G9dB]]
> > >
> > > **Q** [Quality factor vector for each of the NB peaking filters]
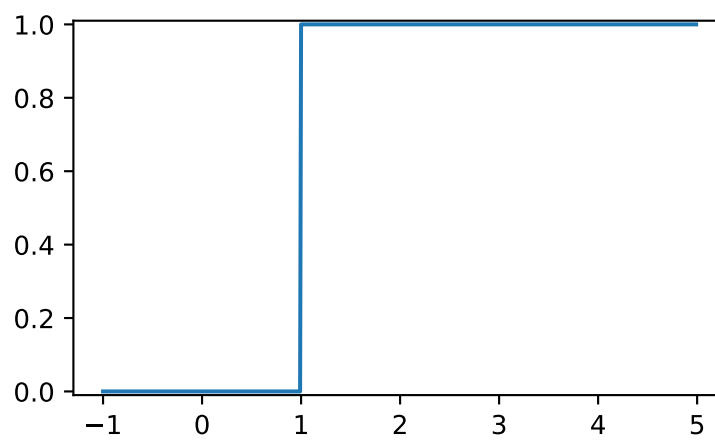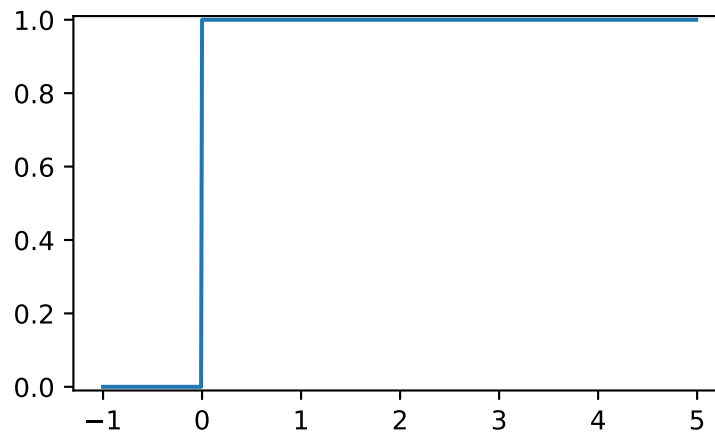
Returns

> **y** [ndarray of output signal samples]

Examples

```
>>> # Test with white noise
>>> w = randn(100000)
>>> y = ten_band_eq_filt(x,GdB)
>>> psd(y,2**10,44.1)
```

sk_dsp_comm.sigsys.**ten_band_eq_resp**(*GdB*, *Q=3.5*)

> Create a frequency response magnitude plot in dB of a ten band equalizer using a semilogplot (semilogx()) type plot
>
> **Parameters**
>
> > **GdB** [Gain vector for 10 peaking filters [G0,...,G9]]
> >
> > **Q** [Quality factor for each peaking filter (default 3.5)]
>
> **Returns**
>
> > **Nothing** [two plots are created]

Examples

```
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm import sigsys as ss
>>> ss.ten_band_eq_resp([0,10.0,0,0,-1,0,5,0,-4,0])
>>> plt.show()
```

sk_dsp_comm.sigsys.**to_wav**(*filename*, *rate*, *x*)

> Write a wave file.
>
> A wrapper function for scipy.io.wavfile.write that also includes int16 scaling and conversion. Assume input x is [-1,1] values.
>
> **Parameters**
>
> > **filename** [file name string]
> >
> > **rate** [sampling frequency in Hz]
>
> **Returns**
>
> > **Nothing** [writes only the *.wav file]
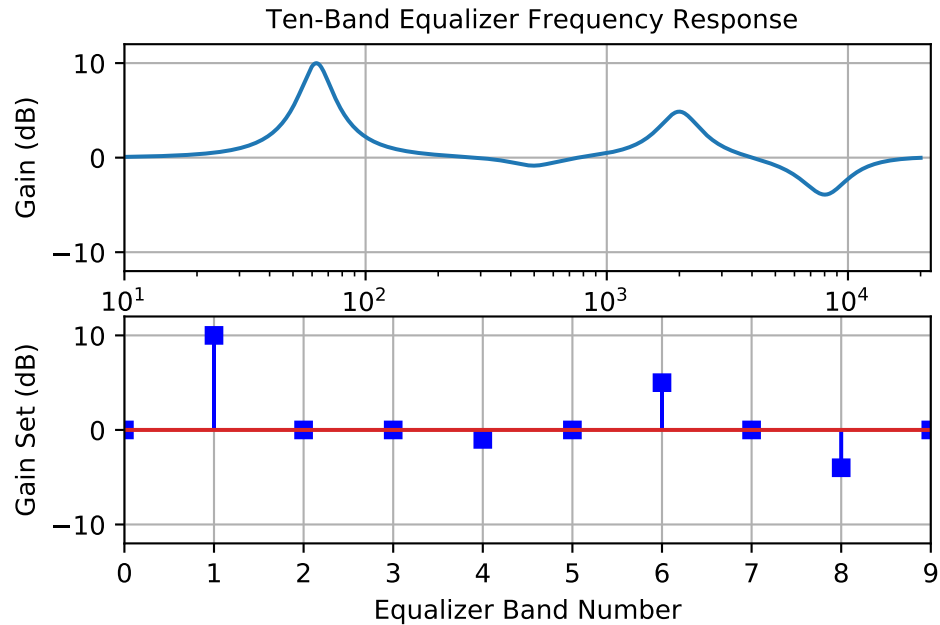
Examples

```
>>> to_wav('test_file.wav', 8000, x)
```

sk_dsp_comm.sigsys.**tri**(*t*, *tau*)

> Approximation to the triangle pulse Lambda(t/tau).
>
> In this numerical version of Lambda(t/tau) the pulse is active over -tau <= t <= tau.
>
> > **Parameters**
> >
> > > **t** [ndarray of the time axis]
> > >
> > > **tau** [one half the triangle base width]
> >
> > **Returns**
> >
> > > **x** [ndarray of the signal Lambda(t/tau)]

Examples

```
>>> import matplotlib.pyplot as plt
>>> from numpy import arange
>>> from sk_dsp_comm.sigsys import tri
>>> t = arange(-1,5,.01)
>>> x = tri(t,1.0)
>>> plt.plot(t,x)
>>> plt.show()
```

To turn on at t = 1, shift t.
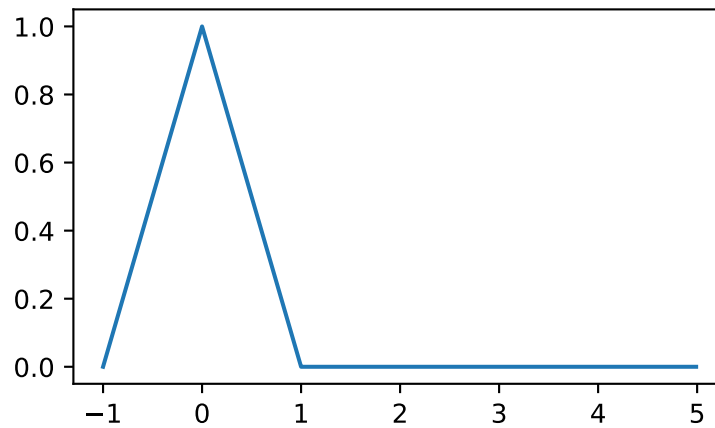
```
>>> x = tri(t - 1.0,1.0)
>>> plt.plot(t,x)
```
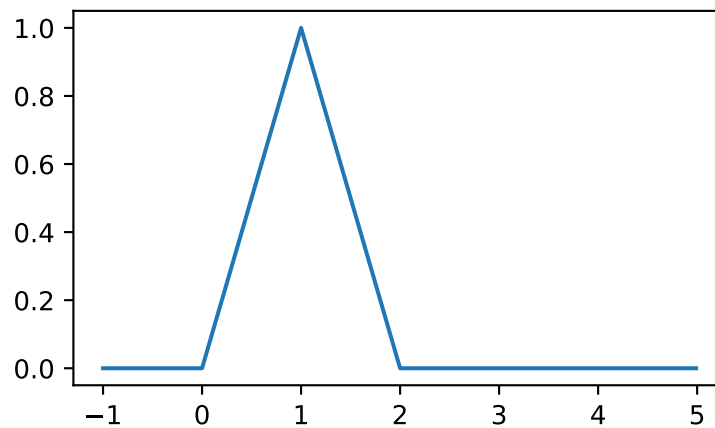

sk_dsp_comm.sigsys.**unique_cpx_roots**(*rlist, tol=0.001*)
   The average of the root values is used when multiplicity is greater than one.

   Mark Wickert October 2016

sk_dsp_comm.sigsys.**upsample**(*x, L*)
   Upsample by factor L

   Insert L - 1 zero samples in between each input sample.

   Parameters

   x [ndarray of input signal values]

L [upsample factor]

Returns

y [ndarray of the output signal values]

Examples

```
>>> y = upsample(x,3)
```

sk_dsp_comm.sigsys.**zplane**(*b*, *a*, *auto_scale=True*, *size=2*, *detect_mult=True*, *tol=0.001*)

Create an z-plane pole-zero plot.

Create an z-plane pole-zero plot using the numerator and denominator z-domain system function coefficient ndarrays b and a respectively. Assume descending powers of z.

Parameters

b [ndarray of the numerator coefficients]

a [ndarray of the denominator coefficients]

auto_scale [bool (default True)]

size [plot radius maximum when scale = False]

Returns

(M,N) [tuple of zero and pole counts + plot window]

Notes

This function tries to identify repeated poles and zeros and will place the multiplicity number above and to the right of the pole or zero. The difficulty is setting the tolerance for this detection. Currently it is set at 1e-3 via the function signal.unique_roots.

Examples

```
>>> # Here the plot is generated using auto_scale
>>> zplane(b,a)
>>> # Here the plot is generated using manual scaling
>>> zplane(b,a,False,1.5)
```

synchronization

A Digital Communications Synchronization and PLLs Function Module

A collection of useful functions when studying PLLs and synchronization and digital comm

Copyright (c) March 2017, Mark Wickert All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

sk_dsp_comm.synchronization.**DD_carrier_sync**(*z*, *M*, *BnTs*, *zeta=0.707*, *mod_type='MPSK'*, *type=0*, *open_loop=False*)

z_prime,a_hat,e_phi = DD_carrier_sync(z,M,BnTs,zeta=0.707,type=0) Decision directed carrier phase tracking

z = complex baseband PSK signal at one sample per symbol M = The PSK modulation order, i.e., 2, 8, or 8.

BnTs = time bandwidth product of loop bandwidth and the symbol period, thus the loop bandwidth as a fraction of the symbol rate.

zeta = loop damping factor type = Phase error detector type: 0 <> ML, 1 <> heuristic

z_prime = phase rotation output (like soft symbol values)

a_hat = the hard decision symbol values landing at the constellation values

e_phi = the phase error e(k) into the loop filter

Ns = Nominal number of samples per symbol (Ts/T) in the carrier phase tracking loop, almost always 1

Kp = The phase detector gain in the carrier phase tracking loop; This value depends upon the algorithm type. For the ML scheme described at the end of notes Chapter 9, A = 1, K 1/sqrt(2), so Kp = sqrt(2).

Mark Wickert July 2014 Updated for improved MPSK performance April 2020 Added experimental MQAM capability April 2020

Motivated by code found in M. Rice, Digital Communications A Discrete-Time Approach, Prentice Hall, New Jersey, 2009. (ISBN 978-0-13-030497-1).

sk_dsp_comm.synchronization.**NDA_symb_sync**(*z*, *Ns*, *L*, *BnTs*, *zeta=0.707*, *I_ord=3*)

z = complex baseband input signal at nominally Ns samples per symbol

Ns = Nominal number of samples per symbol (Ts/T) in the symbol tracking loop, often 4

BnTs = time bandwidth product of loop bandwidth and the symbol period, thus the loop bandwidth as a fraction of the symbol rate.

zeta = loop damping factor

I_ord = interpolator order, 1, 2, or 3

e_tau = the timing error e(k) input to the loop filter

> **Kp = The phase detector gain in the symbol tracking loop; for the** NDA algoithm used
>> here always 1

Mark Wickert July 2014

Motivated by code found in M. Rice, Digital Communications A Discrete-Time Approach, Prentice Hall, New Jersey, 2009. (ISBN 978-0-13-030497-1).

sk_dsp_comm.synchronization.**PLL1**(*theta*, *fs*, *loop_type*, *Kv*, *fn*, *zeta*, *non_lin*)
　　Baseband Analog PLL Simulation Model

> **Parameters**

>> • **theta** – input phase deviation in radians

>> • **fs** – sampling rate in sample per second or Hz

>> • **loop_type** – 1, first-order loop filter F(s)=K_LF; 2, integrator with lead compensation F(s) = (1 + s tau2)/(s tau1), i.e., a type II, or 3, lowpass with lead compensation F(s) = (1 + s tau2)/(1 + s tau1)

>> • **Kv** – VCO gain in Hz/v; note presently assume Kp = 1v/rad and K_LF = 1; the user can easily change this

>> • **fn** – Loop natural frequency (loops 2 & 3) or cutoff frquency (loop 1)

>> • **zeta** – Damping factor for loops 2 & 3

>> • **non_lin** – 0, linear phase detector; 1, sinusoidal phase detector

> **Returns** theta_hat = Output phase estimate of the input theta in radians, ev = VCO control voltage, phi = phase error = theta - theta_hat

Notes

Alternate input in place of natural frequency, fn, in Hz is the noise equivalent bandwidth Bn in Hz.

Mark Wickert, April 2007 for ECE 5625/4625 Modified February 2008 and July 2014 for ECE 5675/4675 Python version August 2014

sk_dsp_comm.synchronization.**PLL_cbb**(*x*, *fs*, *loop_type*, *Kv*, *fn*, *zeta*)
　　Baseband Analog PLL Simulation Model

> **Parameters**

>> • **x** – input phase deviation in radians

>> • **fs** – sampling rate in sample per second or Hz

>> • **loop_type** – 1, first-order loop filter F(s)=K_LF; 2, integrator with lead compensation F(s) = (1 + s tau2)/(s tau1), i.e., a type II, or 3, lowpass with lead compensation F(s) = (1 + s tau2)/(1 + s tau1)

>> • **Kv** – VCO gain in Hz/v; note presently assume Kp = 1v/rad and K_LF = 1; the user can easily change this

>> • **fn** – Loop natural frequency (loops 2 & 3) or cutoff frequency (loop 1)

>> • **zeta** – Damping factor for loops 2 & 3

Returns theta_hat = Output phase estimate of the input theta in radians, ev = VCO control voltage, phi = phase error = theta - theta_hat

Mark Wickert, April 2007 for ECE 5625/4625 Modified February 2008 and July 2014 for ECE 5675/4675 Python version August 2014

sk_dsp_comm.synchronization.**phase_step**(*z*, *ns*, *p_step*, *n_step*)
    Create a one sample per symbol signal containing a phase rotation step Nsymb into the waveform.

> **Parameters**
>
> - **z** – complex baseband signal after matched filter
>
> - **ns** – number of sample per symbol
>
> - **p_step** – size in radians of the phase step
>
> - **n_step** – symbol sample location where the step turns on
>
> **Returns** the one sample symbol signal containing the phase step

Mark Wickert July 2014

sk_dsp_comm.synchronization.**time_step**(*z*, *ns*, *t_step*, *n_step*)
    Create a one sample per symbol signal containing a phase rotation step Nsymb into the waveform.

> **Parameters**
>
> - **z** – complex baseband signal after matched filter
>
> - **ns** – number of sample per symbol
>
> - **t_step** – in samples relative to Ns
>
> - **n_step** – symbol sample location where the step turns on
>
> **Returns** the one sample per symbol signal containing the phase step

Mark Wickert July 2014

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

X

Z