scikit-dsp-comm Documentation

Release v0.0.4

Mark Wickert, Chiranth Siddappa

May 07, 2018

Modules

CHAPTER 1

Examples

- SciPy 2017 Tutorial
- Jupyter Notebook Examples

1.1 coeff2header

Digital Filter Coefficient Conversion to C Header Files

`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.`**`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

`sk_dsp_comm.coeff2header.`**`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))

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

Npts [number of points to plot; default is 1024]

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

Mark Wickert, January 2015

1.2 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_chan`**(*x_bits*, *EBN0_dB*)

Parameters

x_bits [serial bit stream of 0/1 values.]

EBNO_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.`**`BPSK_BEP`**(*tx_data*, *rx_data*, *Ncorr=1024*, *Ntransient=0*)

Count bit errors between a transmitted and received BPSK signal. Time delay between streams is detected as well as ambiquity 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,'src')
```

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.**MPSK_bb**(*N_symb*, *Ns*, *M*, *pulse='rect'*, *alpha=0.25*, *MM=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,]

M [MPSK modulation order, e.g., 4, 8, 16, . . .]

pulse_type [‘rect’ , ‘rc’, ‘src’ (default ‘rect’)]

alpha [excess bandwidth factor(default 0.25)]

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

Returns

x [ndarray of the MPSK signal values]

b [ndarray of the pulse shape]

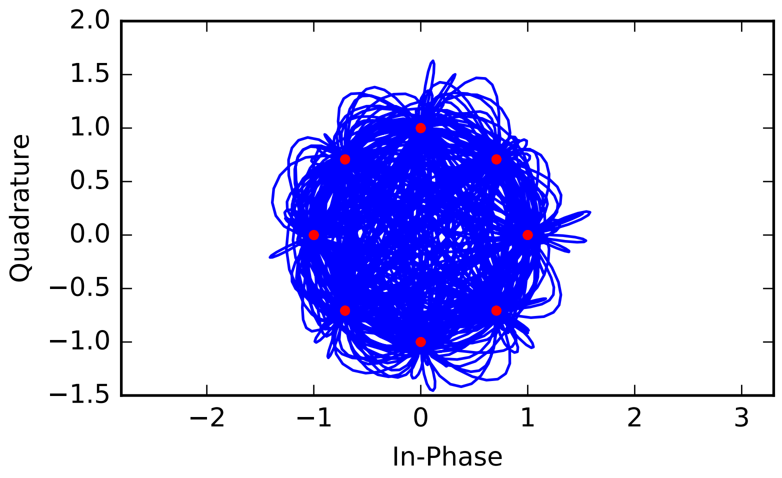
data [ndarray of the underlying data bits]

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.`**`OFDM_rx`**(*x*, *Nf*, *N*, *Np=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]

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

Np [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.]

nt [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.]

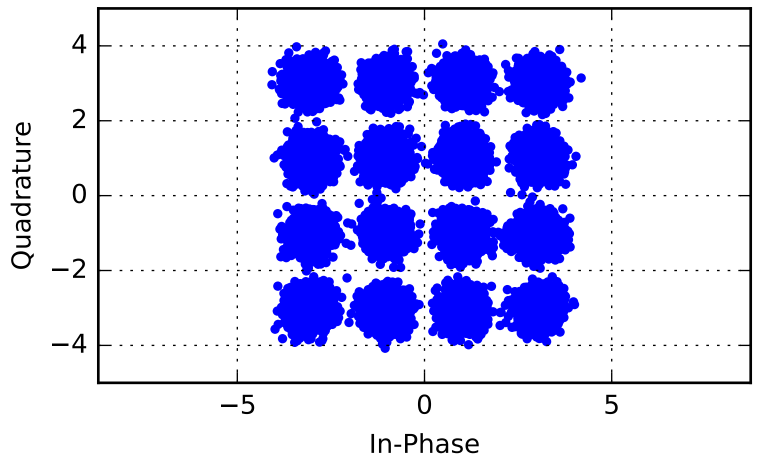
H [channel estimate (in the frequency domain at each subcarrier)]

See also:

OFDM_tx

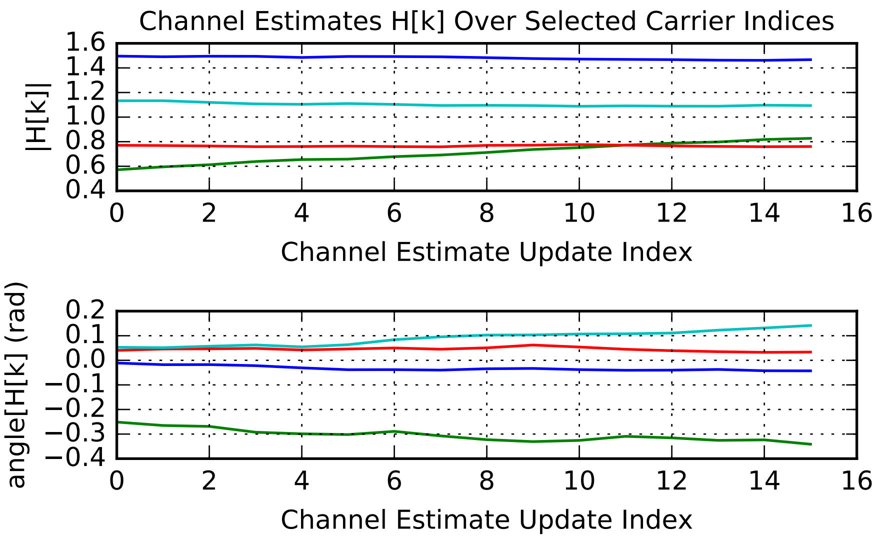
Examples

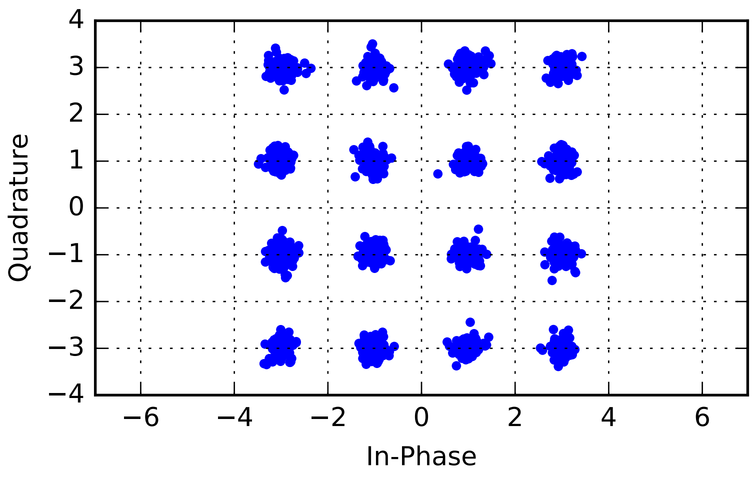
```
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm import digitalcom as dc
>>> from scipy import signal
>>> from numpy import array
>>> hc = array([1.0, 0.1, -0.05, 0.15, 0.2, 0.05]) # impulse response spanning␣
↪five symbols
>>> # Quick example using the above channel with no cyclic prefix
>>> x1,b1,IQ_data1 = dc.QAM_bb(50000,1,'16qam')
>>> x_out = dc.OFDM_tx(IQ_data1,32,64,0,True,0)
>>> 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')
>>> plt.axis('equal')
>>> plt.grid()
>>> plt.show()
```





sk_dsp_comm.digitalcom.**OFDM_tx**(*IQ_data*, *Nf*, *N*, *Np=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]

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

Np [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

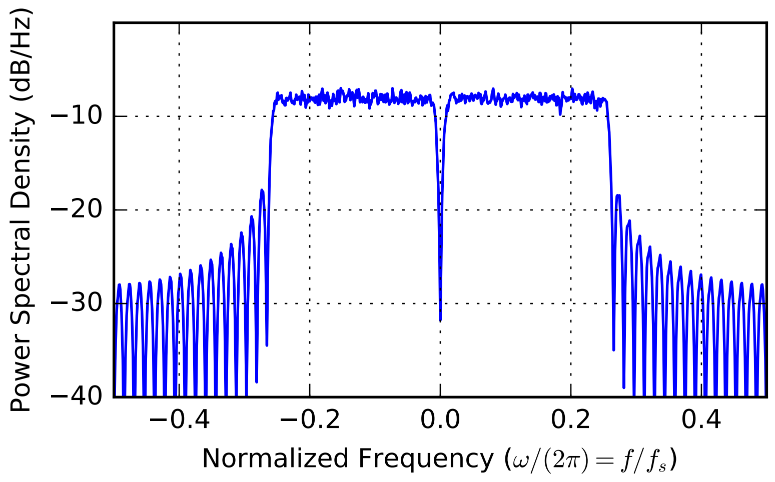
x_out [complex baseband OFDM waveform output after P/S and CP insertion]

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)
>>> 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*)

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 xhat = decoded PCM signal samples

Mark Wickert, March 2015

`sk_dsp_comm.digitalcom.`**`PCM_encode`**(*x*, *N_bits*)

x = signal samples to be PCM encoded

N_bits = bit precision of PCM samples x_bits = encoded serial bit stream of 0/1 values. MSB first. /// Mark Wickert, Mark 2015

`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.`**`QAM_bb`**(*N_symb*, *Ns*, *mod_type='16qam'*, *pulse='rect'*, *alpha=0.35*)
QAM_BB_TX: A complex baseband transmitter x,b,tx_data = QAM_bb(K,Ns,M)

//////////// Inputs //

N_symb = the number of symbols to process Ns = number of samples per symbol

mod_type = 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.

SRC = pulse shape: 0-> rect, 1-> SRC

//////////// Outputs //

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.`**`QPSK_BEP`**(*tx_data*, *rx_data*, *Ncorr=1024*, *Ntransient=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_bb`**(*N_symb*, *Ns*, *lfsr_len=5*, *pulse='src'*, *alpha=0.25*, *M=6*)

`sk_dsp_comm.digitalcom.`**`QPSK_rx`**(*fc*, *N_symb*, *Rs*, *EsN0=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.`**`Q_fctn`**(*x*)

Gaussian Q-function

`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_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 RZ signal values]

b [ndarray of the pulse shape]

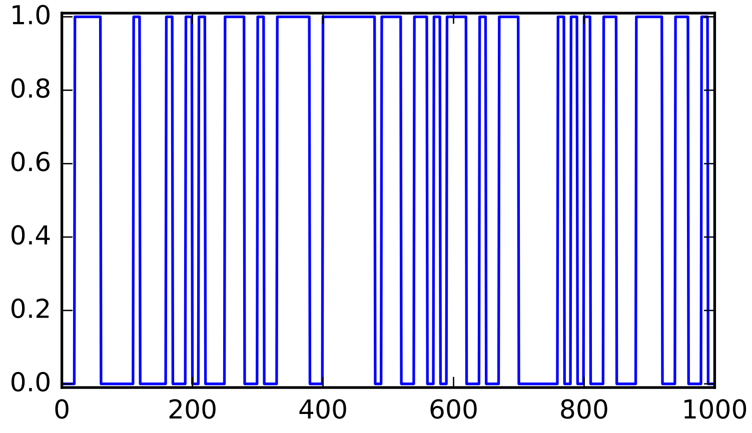
data [ndarray of the underlying data bits]

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.`**`bit_errors`**(*tx_data*, *rx_data*, *Ncorr=1024*, *Ntransient=0*)

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

`sk_dsp_comm.digitalcom.`**`chan_est_equalize`**(*z*, *Np*, *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.]

Np [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,Np,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]

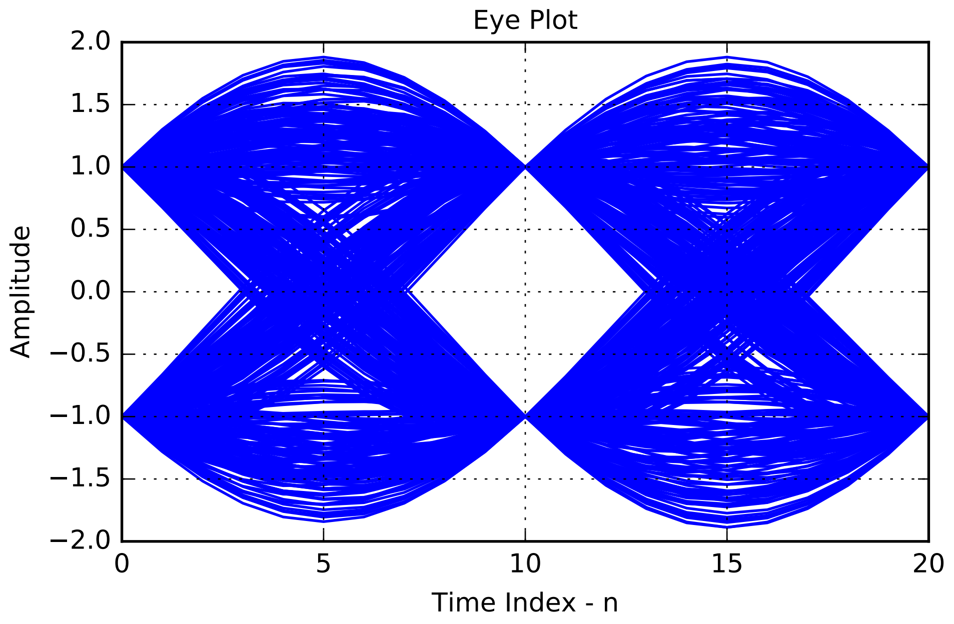
Notes

Increase S to eliminate filter transients.

Examples

1000 bits at 10 samples per bit with ‘rc’ shaping.

```
>>> 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.

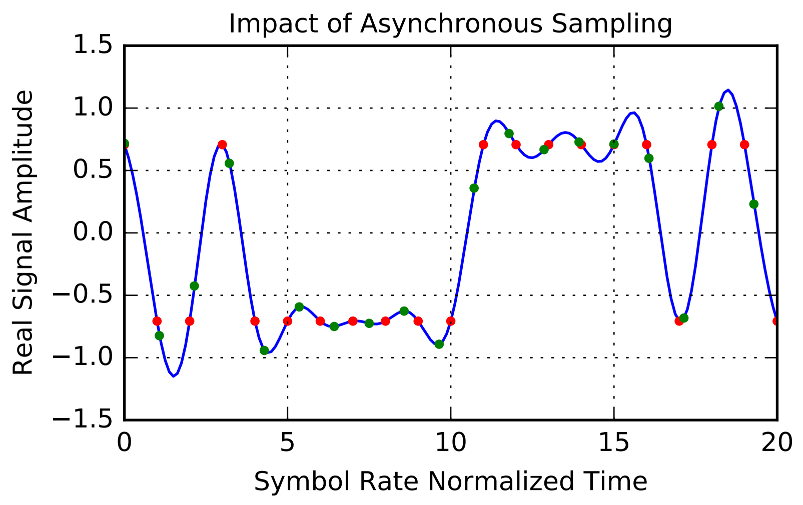
A founding paper on the subject of interpolators is: C. W. Farrow, "A Continuously variable Digital Delay Element," Proceedings of the IEEE Intern. Symp. on Circuits Syst., pp. 2641-2645, June 1988.

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.**mux_pilot_blocks**(*IQ_data*, *Np*)

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

Np [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]

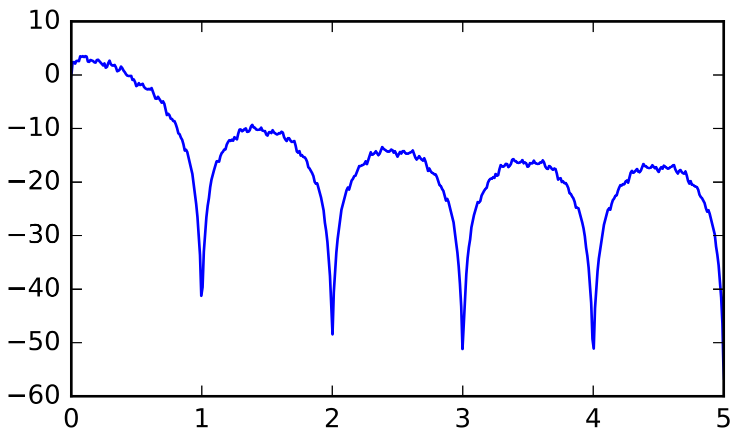
f [ndarray of frequency values]

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

```
>>> 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.`**`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

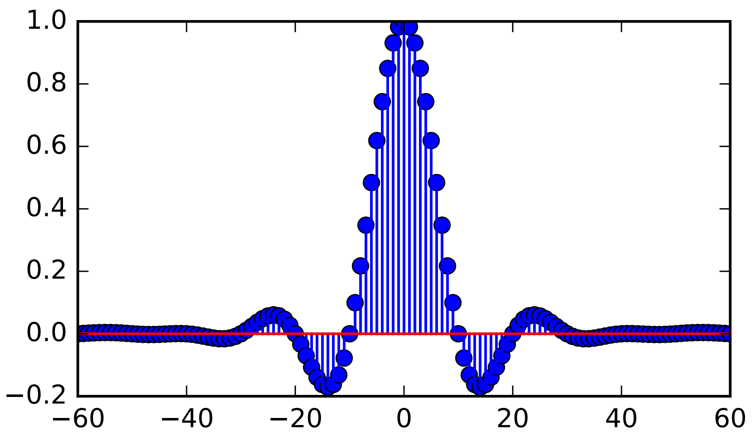
Notes

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.`**`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

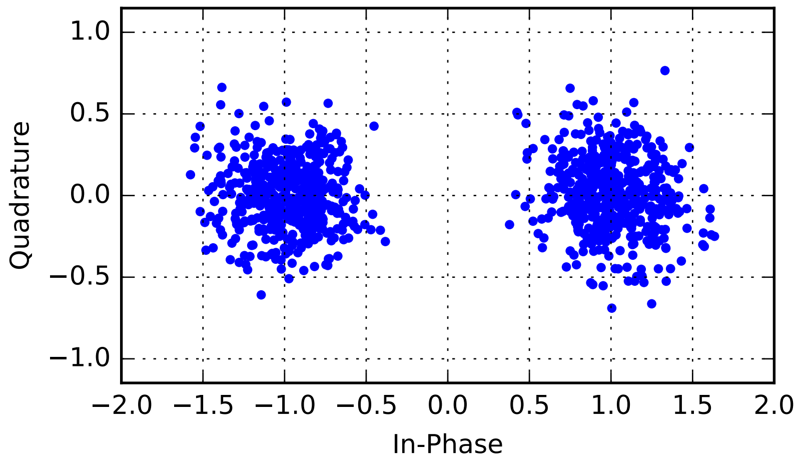
Normally the signal is complex, so the scatter plot contains clusters at point in the complex plane. For a binary signal such as BPSK, the point centers are nominally +/-1 on the real axis. Start is used to eliminate transients from the FIR pulse shaping filters from appearing in the scatter plot.

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]

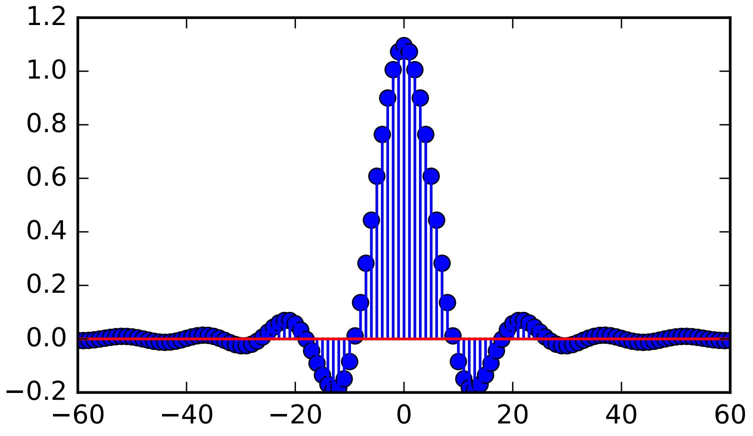
Notes

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.**tobin**(*data*, *width*)

sk_dsp_comm.digitalcom.**xcorr**(*x1*, *x2*, *Nlags*)

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

1.3 fec_conv

A Convolutional Encoding and Decoding

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

`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

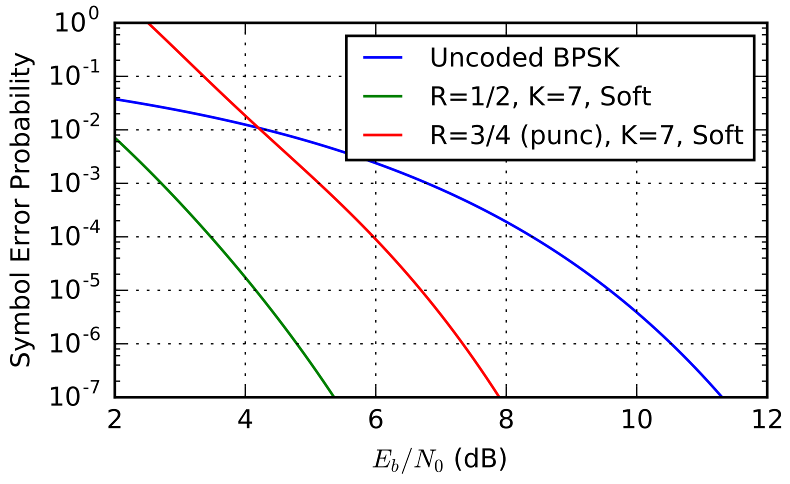
M: M-ary

Notes

The code rate R is given by $R_s = \frac{k}{n}$.

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()
```



class sk_dsp_comm.fec_conv.**fec_conv**(*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
>>> cc1 = fec_conv.fec_conv(('101', '111'), Depth=10)  # decision depth is 10
```

Methods

`bm_calc`(ref_code_bits, rec_code_bits, . . .)	distance = bm_calc(ref_code_bits, rec_code_bits, metric_type)
`conv_encoder`(input, state)	output, state = conv_encoder(input,state)
`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])	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*)

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

Mark Wickert February 2014

conv_encoder (*input*, *state*)

output, state = conv_encoder(input,state) We assume a rate 1/2 encoder. 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 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 February 2014

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 fec_conv
>>> cc = fec_conv(('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 fec_conv
>>> cc = fec_conv(('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)*)

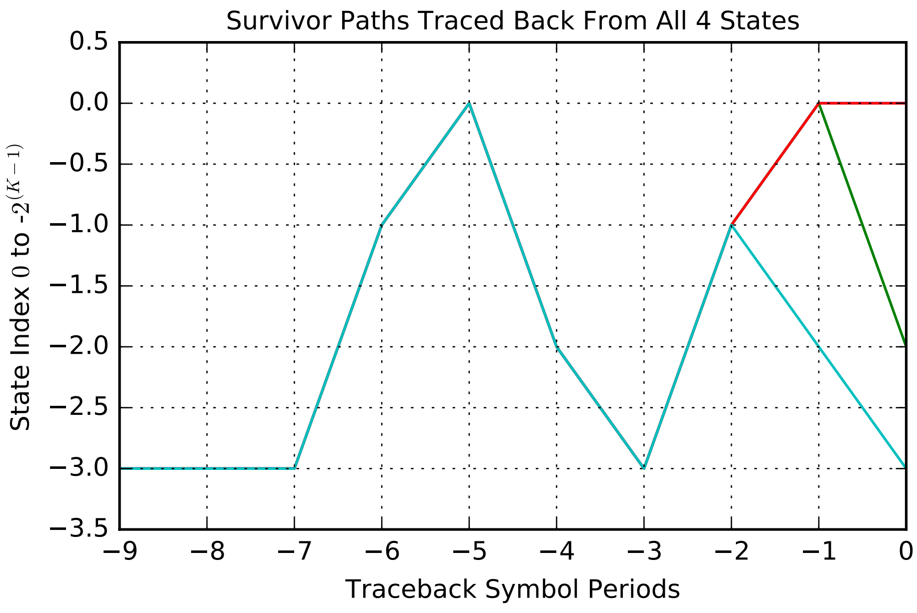
Plots a path of the possible last 4 states.

Parameters

fsize [Plot size for matplotlib.]

Examples

```
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm.fec_conv import fec_conv
>>> from sk_dsp_comm import digitalcom as dc
>>> import numpy as np
>>> cc = fec_conv()
>>> 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)*)

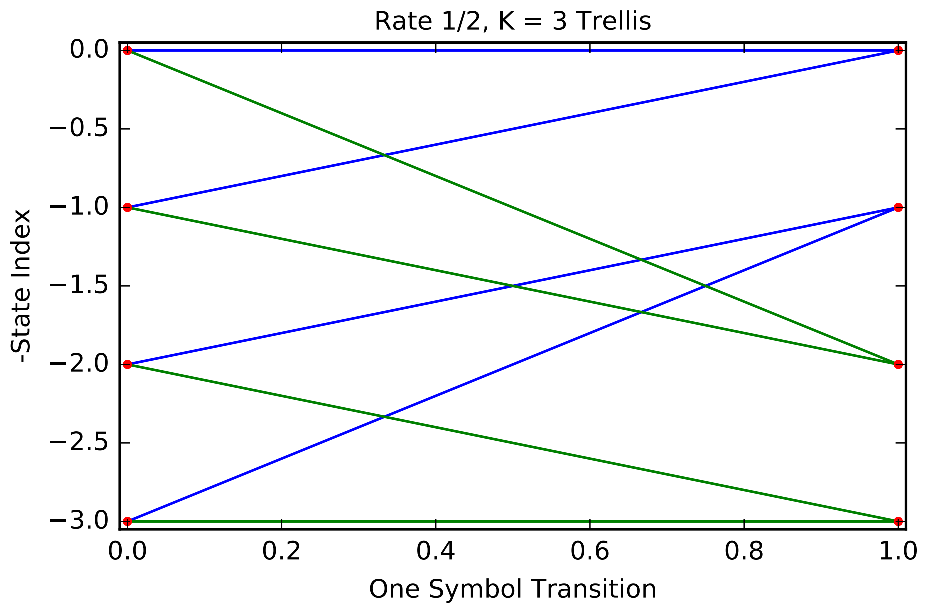
Plots a trellis diagram of the possible state transitions.

Parameters

fsize [Plot size for matplotlib.]

Examples

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



viterbi_decoder (*x*, *metric_type='three_bit'*)

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 or soft decision decoding type. At present only 3-bit soft-decision is implemented.

Returns

y: Decoded 0/1 bit stream

Examples

Take from fall 2016 final project

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

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

Mark Wickert November 2014

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

class sk_dsp_comm.fec_conv.**trellis_branches** (*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.**trellis_nodes**(*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.**trellis_paths**(*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.

1.4 fir_design_helper

Basic Linear Phase Digital Filter Design Helper

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*)
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

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*)
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

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

`sk_dsp_comm.fir_design_helper.`**`fir_remez_lpf`**(*f_pass*, *f_stop*, *d_pass*, *d_stop*, *fs=1.0*, *N_bump=5*)

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

`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*)

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*)

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*)

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*)

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*, *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

1.5 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'*)

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'*)

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'*)

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'*)

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*, *Npts=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

1.6 multirate_helper

`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; deflut is 1024

fsize = figure size; deflut 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

1.7 optfir

Routines for designing optimal FIR filters.

For a great intro to how all this stuff works, see section 6.6 of "Digital Signal Processing: A Practical Approach", Emmanuael C. Ifeachor and Barrie W. Jervis, Adison-Wesley, 1993. ISBN 0-201-54413-X.

sk_dsp_comm.optfir.**bporder** (*freq1*, *freq2*, *delta_p*, *delta_s*)
FIR bandpass filter length estimator. freq1 and freq2 are normalized to the sampling frequency. delta_p is the passband deviation (ripple), delta_s is the stopband deviation (ripple).

From Mintzer and Liu (1979)

sk_dsp_comm.optfir.**lporder** (*freq1*, *freq2*, *delta_p*, *delta_s*)
FIR lowpass filter length estimator. freq1 and freq2 are normalized to the sampling frequency. delta_p is the passband deviation (ripple), delta_s is the stopband deviation (ripple).

Note, this works for high pass filters too (freq1 > freq2), but doesnt work well if the transition is near f == 0 or f == fs/2

From Herrmann et al (1973), Practical design rules for optimum finite impulse response filters. Bell System Technical J., 52, 769-99

sk_dsp_comm.optfir.**passband_ripple_to_dev** (*ripple_db*)
Convert passband ripple spec expressed in dB to an absolute value

sk_dsp_comm.optfir.**remezord** (*fcuts*, *mags*, *devs*, *fsamp=2*)
FIR order estimator (lowpass, highpass, bandpass, mulitiband).

(n, fo, ao, w) = remezord (f, a, dev) (n, fo, ao, w) = remezord (f, a, dev, fs)

(n, fo, ao, w) = remezord (f, a, dev) finds the approximate order, normalized frequency band edges, frequency band amplitudes, and weights that meet input specifications f, a, and dev, to use with the remez command.

- f is a sequence of frequency band edges (between 0 and Fs/2, where Fs is the sampling frequency), and a is a sequence specifying the desired amplitude on the bands defined by f. The length of f is twice the length of a, minus 2. The desired function is piecewise constant.
- dev is a sequence the same size as a that specifies the maximum allowable deviation or ripples between the frequency response and the desired amplitude of the output filter, for each band.

Use remez with the resulting order n, frequency sequence fo, amplitude response sequence ao, and weights w to design the filter b which approximately meets the specifications given by remezord input parameters f, a, and dev:

b = remez (n, fo, ao, w)

(n, fo, ao, w) = remezord (f, a, dev, Fs) specifies a sampling frequency Fs.

Fs defaults to 2 Hz, implying a Nyquist frequency of 1 Hz. You can therefore specify band edges scaled to a particular applications sampling frequency.

In some cases remezord underestimates the order n. If the filter does not meet the specifications, try a higher order such as n+1 or n+2.

`sk_dsp_comm.optfir.`**`stopband_atten_to_dev`**(*atten_db*)

Convert a stopband attenuation in dB to an absolute value

1.8 pyaudio_helper

Support functions and classes for using PyAudio for real-time DSP

`class` `sk_dsp_comm.pyaudio_helper.`**`DSP_io_stream`**(*stream_callback*, *in_idx=1*, *out_idx=4*, *frame_length=1024*, *fs=44100*, *Tcapture=0*, *sleep_time=0.1*)

Real-time DSP one channel input/output audio streaming

Use PyAudio to explore real-time audio DSP using Python

Mark Wickert, Andrew Smit September 2017

Methods

`DSP_callback_tic`()	Add new tic time to the DSP_tic list.
`DSP_callback_toc`()	Add new toc time to the DSP_toc list.
`DSP_capture_add_samples`(new_data)	Append new samples to the data_capture array and increment the sample counter If length reaches Tcapture, then the newest samples will be kept.
`DSP_capture_add_samples_stereo`(…)	Append new samples to the data_capture_left array and the data_capture_right array and increment the sample counter.
`cb_active_plot`(start_ms, stop_ms[, line_color])	Plot timing information of time spent in the callback.
`get_LR`(in_data)	Splits incoming packed stereo data into separate left and right channels
`in_out_check`()	Checks the input and output to see if they are valid
`interactive_stream`([Tsec, numChan])	Stream audio with start and stop radio buttons
`pack_LR`(left_out, right_out)	Packs separate left and right channel data into one array to output and returns the output.
`stop`()	Call to stop streaming
`stream`([Tsec, numChan])	Stream audio using callback
`stream_stats`()	Display basic statistics of callback execution: ideal period between callbacks, average measured period between callbacks, and average time spent in the callback.
`thread_stream`([Tsec, numChan])	Stream audio in a thread using callback.

interaction	

DSP_callback_tic()
Add new tic time to the DSP_tic list. Will not be called if Tcapture = 0.

DSP_callback_toc()
Add new toc time to the DSP_toc list. Will not be called if Tcapture = 0.

DSP_capture_add_samples(*new_data*)
Append new samples to the data_capture array and increment the sample counter If length reaches Tcapture, then the newest samples will be kept. If Tcapture = 0 then new values are not appended to the data_capture array.

DSP_capture_add_samples_stereo(*new_data_left*, *new_data_right*)
Append new samples to the data_capture_left array and the data_capture_right array and increment the sample counter. If length reaches Tcapture, then the newest samples will be kept. If Tcapture = 0 then new values are not appended to the data_capture array.

cb_active_plot(*start_ms*, *stop_ms*, *line_color='b'*)
Plot timing information of time spent in the callback. This is similar to what a logic analyzer provides when probing an interrupt.

cb_active_plot(start_ms,stop_ms,line_color='b')

get_LR(*in_data*)
Splits incoming packed stereo data into separate left and right channels and returns an array of left samples and an array of right samples

Parameters

in_data [input data from the streaming object in the callback function.]

Returns

left_in [array of incoming left channel samples]

right_in [array of incoming right channel samples]

`in_out_check`()
Checks the input and output to see if they are valid

`interactive_stream`(*Tsec=2*, *numChan=1*)
Stream audio with start and stop radio buttons

Interactive stream is designed for streaming audio through this object using a callback function. This stream is threaded, so it can be used with ipywidgets. Click on the "Start Streaming" button to start streaming and click on "Stop Streaming" button to stop streaming.

Parameters

Tsec [stream time in seconds if Tsec > 0. If Tsec = 0, then stream goes to infinite]

mode. When in infinite mode, the "Stop Streaming" radio button or Tsec.stop() can be

used to stop the stream.

numChan [number of channels. Use 1 for mono and 2 for stereo.]

`pack_LR`(*left_out*, *right_out*)
Packs separate left and right channel data into one array to output and returns the output.

Parameters

left_out [left channel array of samples going to output]

right_out [right channel array of samples going to output]

Returns

out [packed left and right channel array of samples]

`stop`()
Call to stop streaming

`stream`(*Tsec=2*, *numChan=1*)
Stream audio using callback

Parameters

Tsec [stream time in seconds if Tsec > 0. If Tsec = 0, then stream goes to infinite]

mode. When in infinite mode, Tsec.stop() can be used to stop the stream.

numChan [number of channels. Use 1 for mono and 2 for stereo.]

`stream_stats`()
Display basic statistics of callback execution: ideal period between callbacks, average measured period between callbacks, and average time spent in the callback.

`thread_stream`(*Tsec=2*, *numChan=1*)
Stream audio in a thread using callback. The stream is threaded, so widgets can be used simultaneously during stream.

Parameters

Tsec [stream time in seconds if Tsec > 0. If Tsec = 0, then stream goes to infinite]

mode. When in infinite mode, Tsec.stop() can be used to stop the stream.

numChan [number of channels. Use 1 for mono and 2 for stereo.]

`sk_dsp_comm.pyaudio_helper.`**`available_devices`**`()`
  Display available input and output audio devices along with their port indices.

`class` `sk_dsp_comm.pyaudio_helper.`**`loop_audio`**`(`*x*, *start_offset=0*`)`
  Loop signal ndarray during playback. Optionally start_offset samples into the array.

  Mark Wickert July 2017

Methods

get_samples	

`get_samples``(`*frame_count*`)`

1.9 rtlsdr_helper

Support functions for the RTL-SDR using pyrtlsdr

Copyright (c) July 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.

`sk_dsp_comm.rtlsdr_helper.`**`complex2wav`**`(`*filename*, *rate*, *x*`)`
  Save a complex signal vector to a wav file for compact binary storage of 16-bit signal samples. The wav left and right channels are used to save real (I) and imaginary (Q) values. The rate is just a convent way of documenting the original signal sample rate.

  complex2wav(filename,rate,x)

  Mark Wickert April 2014

`sk_dsp_comm.rtlsdr_helper.`**`discrim`**`(`*x*`)`
  function disdata = discrim(x) where x is an angle modulated signal in complex baseband form.

  Mark Wickert

`sk_dsp_comm.rtlsdr_helper.`**`fsk_BEP`**(*rx_data*, *m*, *flip*)
Estimate the BEP of the data bits recovered by the RTL-SDR Based FSK Receiver.

The reference m-sequence generated in Python was found to produce sequences running in the opposite direction relative to the m-sequences generated by the mbed. To allow error detection the reference m-sequence is flipped.

Mark Wickert April 2014

`sk_dsp_comm.rtlsdr_helper.`**`mono_FM`**(*x*, *fs=2400000.0*, *file_name='test.wav'*)
Decimate complex baseband input by 10 Design 1st decimation lowpass filter (f_c = 200 KHz)

`sk_dsp_comm.rtlsdr_helper.`**`pilot_PLL`**(*xr*, *fq*, *fs*, *loop_type*, *Bn*, *zeta*)
Mark Wickert, April 2014

`sk_dsp_comm.rtlsdr_helper.`**`sccs_bit_sync`**(*y*, *Ns*)

Symbol synchronization algorithm using SCCS

y = baseband NRZ data waveform

Ns = nominal number of samples per symbol

Reworked from ECE 5675 Project Translated from m-code version Mark Wickert April 2014

`sk_dsp_comm.rtlsdr_helper.`**`stereo_FM`**(*x*, *fs=2400000.0*, *file_name='test.wav'*)
Stereo demod from complex baseband at sampling rate fs. Assume fs is 2400 ksps

Mark Wickert July 2017

`sk_dsp_comm.rtlsdr_helper.`**`wav2complex`**(*filename*)
Return a complex signal vector from a wav file that was used to store the real (I) and imaginary (Q) values of a complex signal ndarray. The rate is included as means of recalling the original signal sample rate.

fs,x = wav2complex(filename)

Mark Wickert April 2014

1.10 sigsys

Signals and Systems Function Module

`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.`**`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.`**`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]

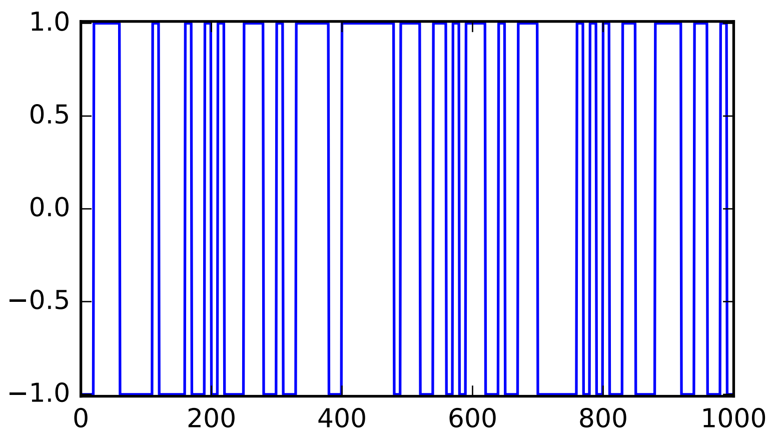
data [ndarray of the underlying data bits]

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]

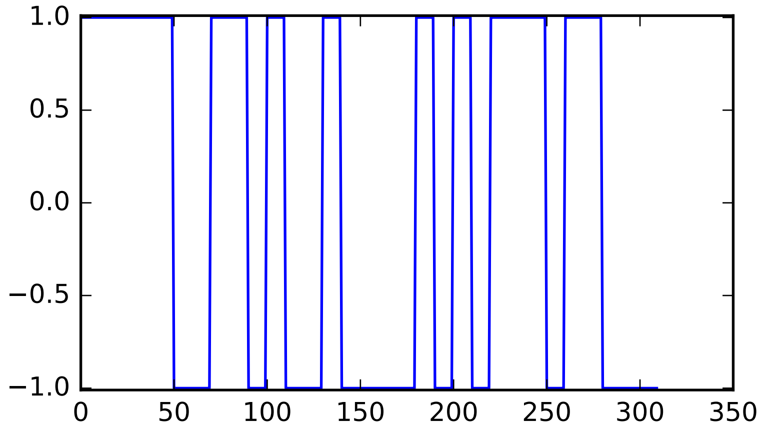
b [ndarray of the pulse shape]

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*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

```
>>> 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.**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.**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]

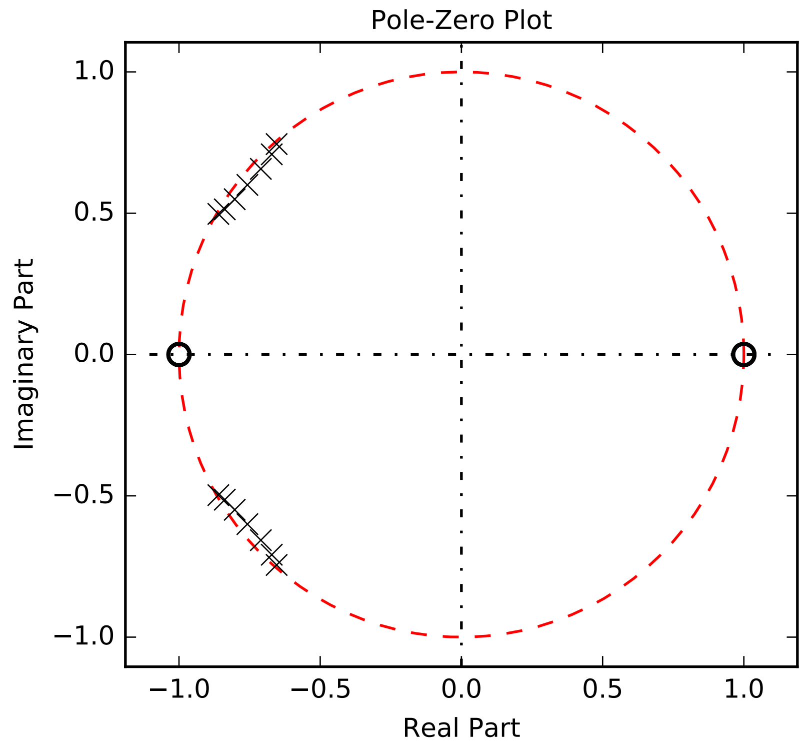
a_bpf [ndarray of the denominator filter coefficients]

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()
```

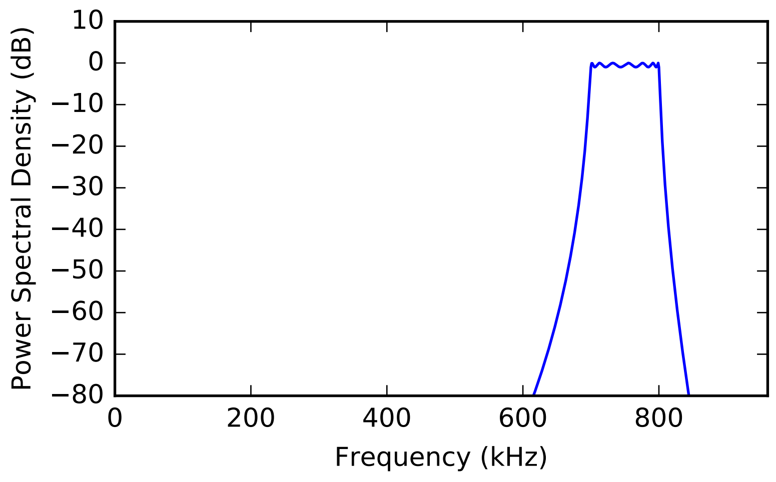
Pole-zero plot of the filter.

```
>>> 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.`**`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

```
>>> 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.`**`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.`**`conv_integral`**(*x1*, *tx1*, *x2*, *tx2*, *extent=('f', 'f')*)

Continuous-time convolution of x1 and x2 with proper tracking of the output time axis.

Appromimate 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]

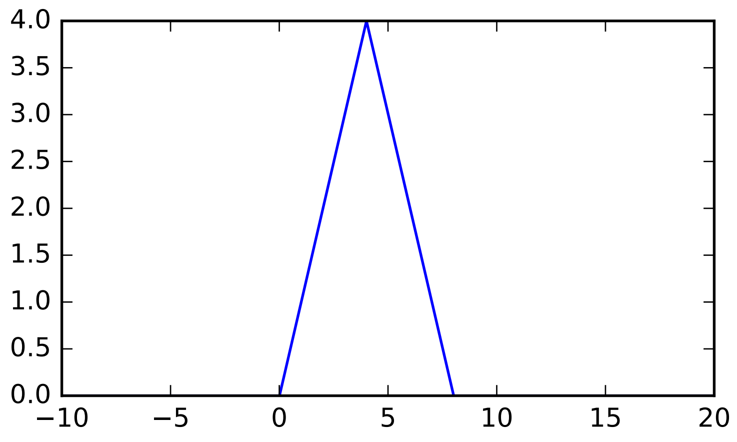
ty [ndarray of the corresponding time axis for y]

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.

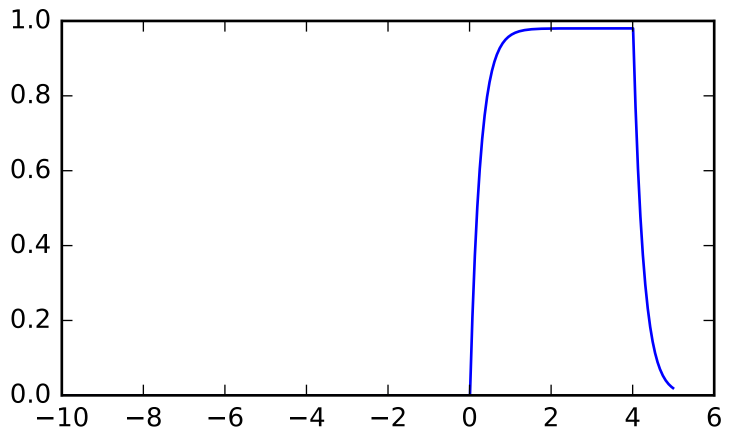
Examples

```
>>> 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]

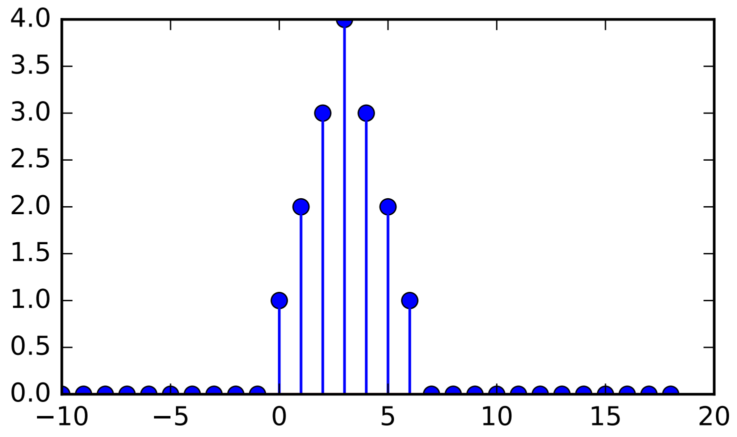
ny [ndarray of the corresponding sequence index n]

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.

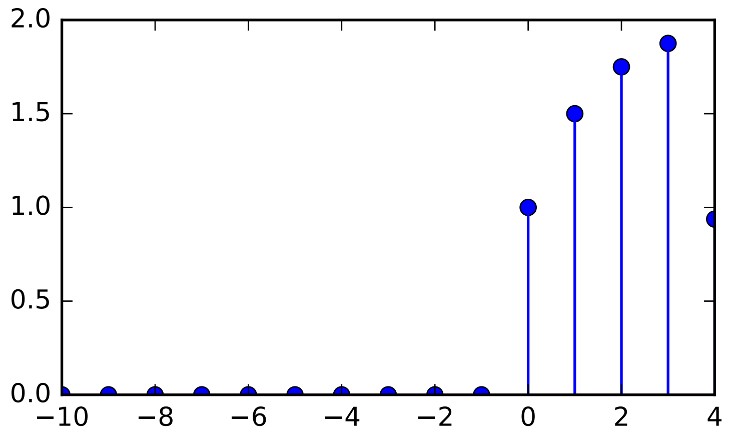
Examples

```
>>> 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*, *EsN0*, *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 natuarl 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]

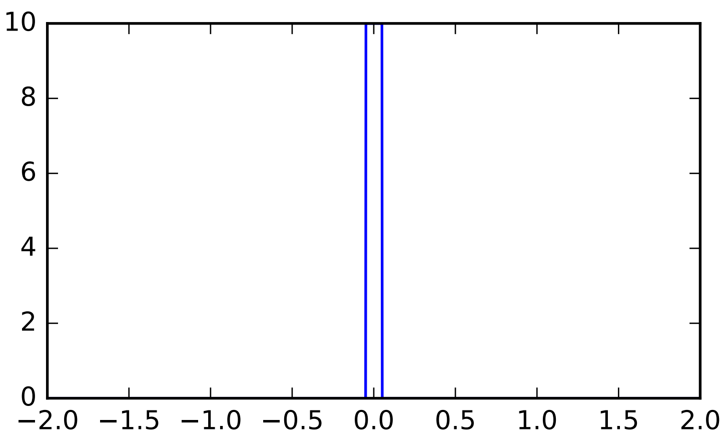
eps [pulse width]

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

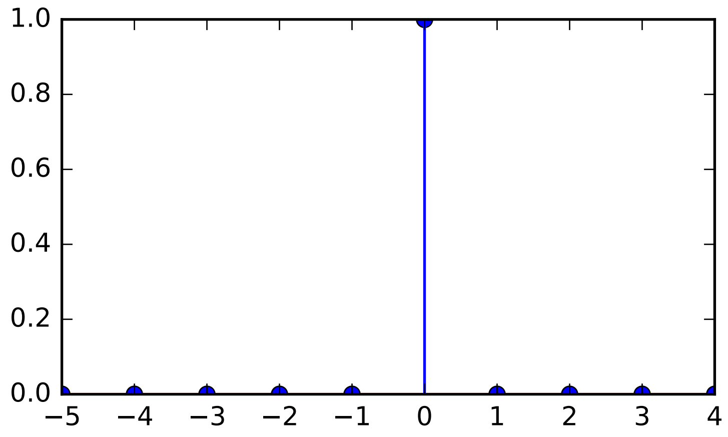
n [ndarray of the time axis]

Returns

x [ndarray of the signal delta[n]]

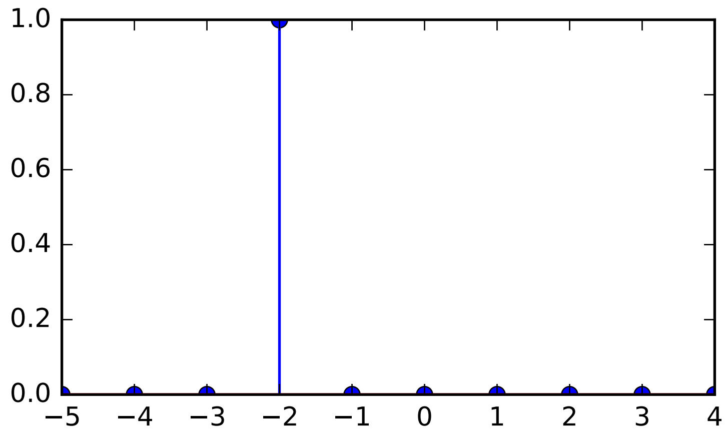
Examples

```
>>> 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 [upsample 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.**direct**(*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

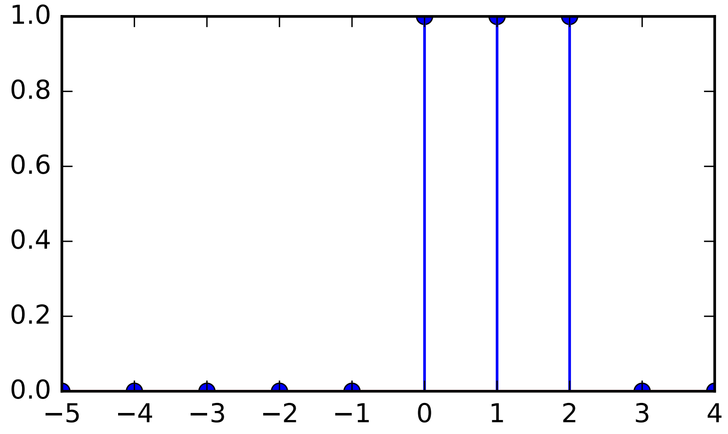
x [ndarray of the signal]

Notes

The discrete rectangle turns on at n = 0, off at n = N-1 and has duration of exactly N samples.

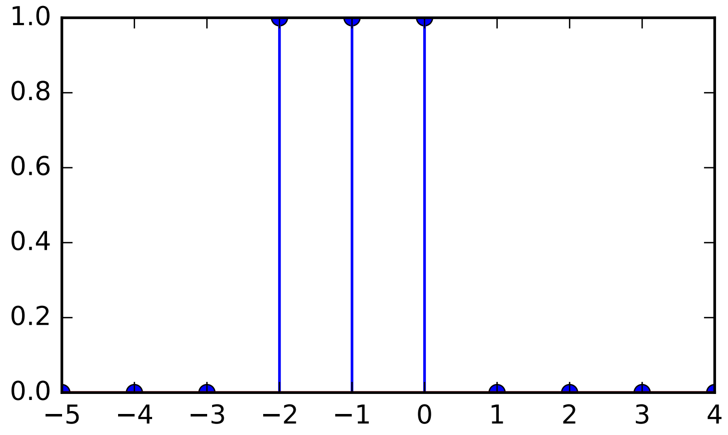
Examples

```
>>> 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

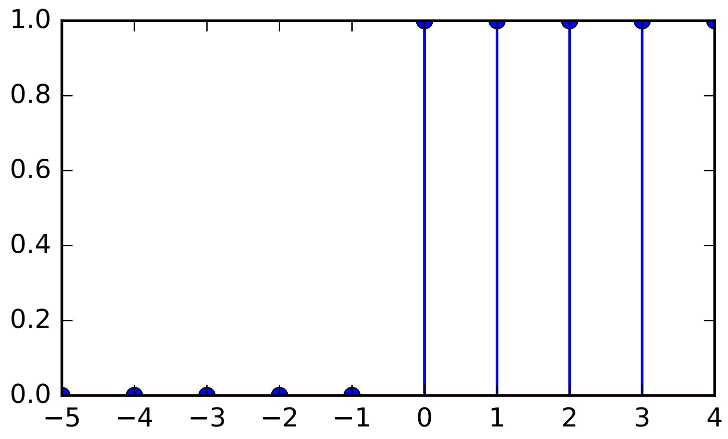
n [ndarray of the time axis]

Returns

x [ndarray of the signal u[n]]

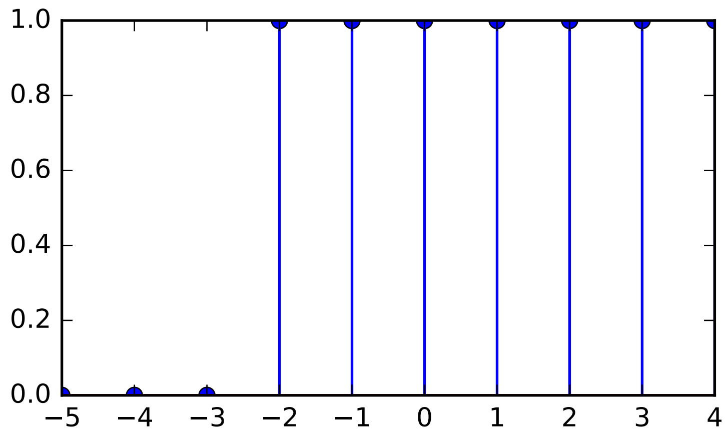
Examples

```
>>> 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

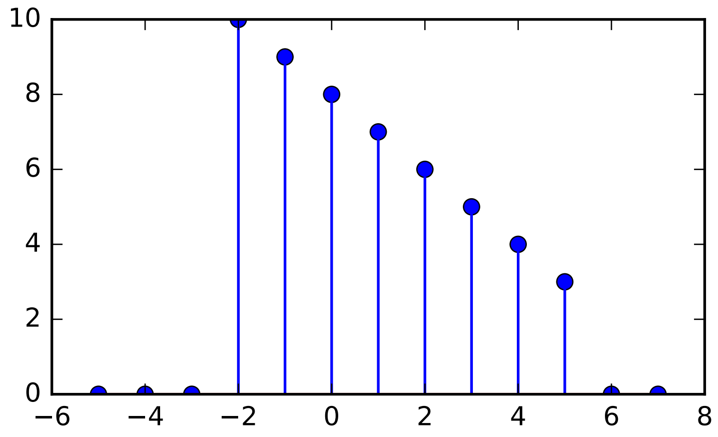
n [time index ndarray covering at least -2 to +5.]

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]

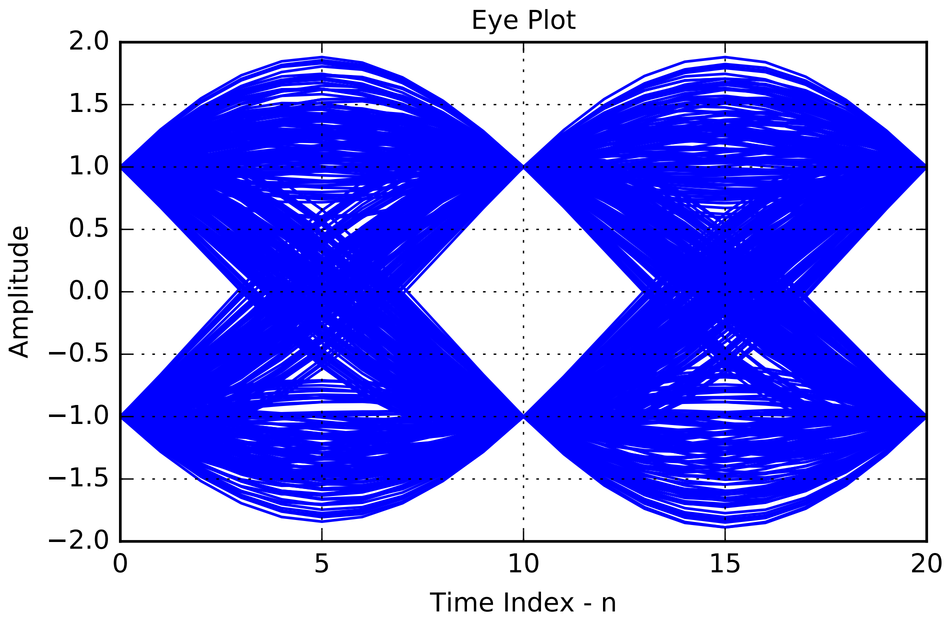
Notes

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]

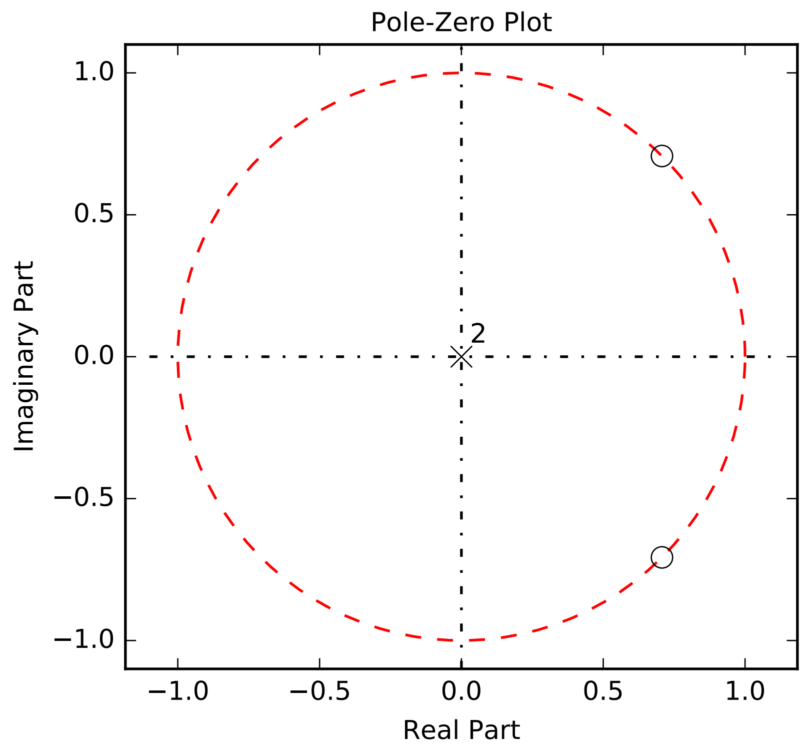
a [denominator coefficient ndarray]

Notes

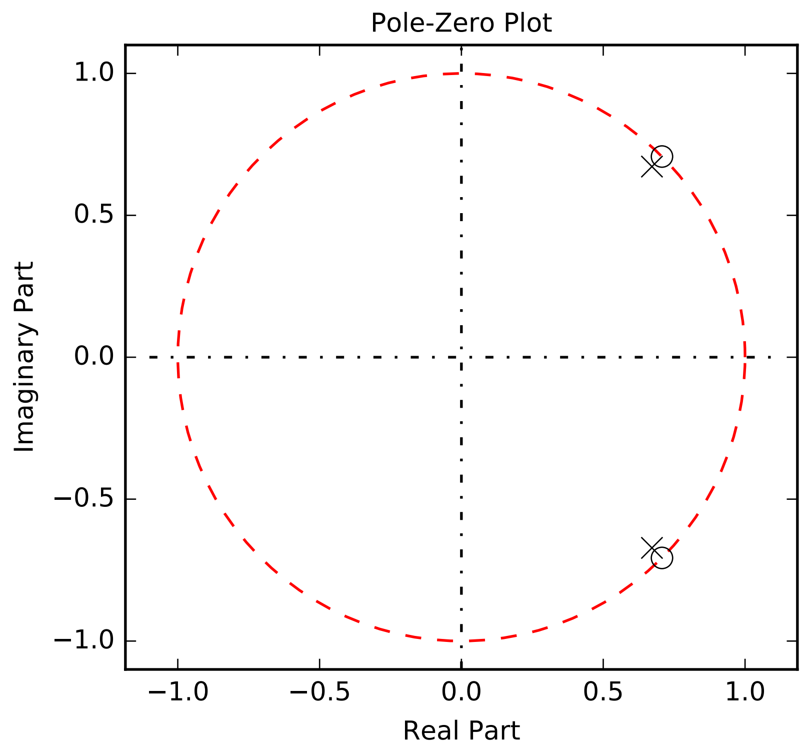
If the pole radius is 0 then an FIR version is created, that is there are no poles except at z = 0.

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 assummed 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]]

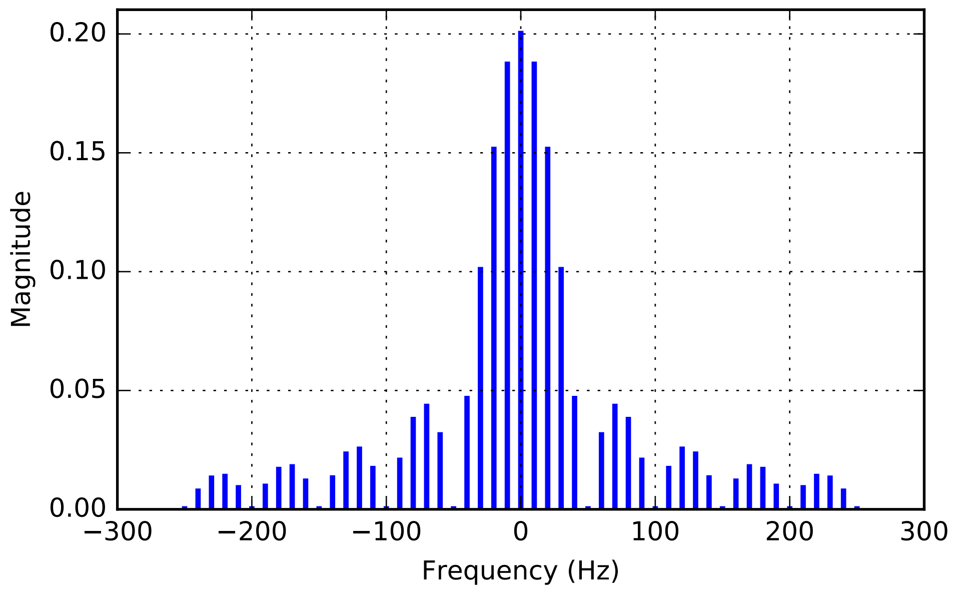
fk [ndarray of the harmonic frequencies [0, f0,2f0,. . . ,Nf0]]

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]

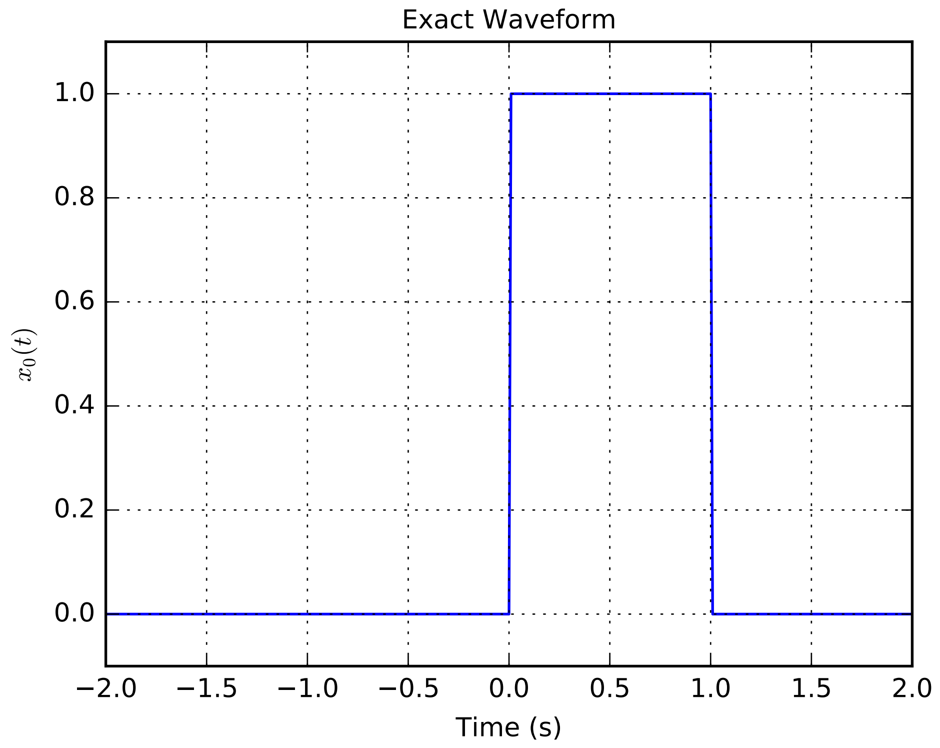
X [the Fourier transform approximation (complex)]

Notes

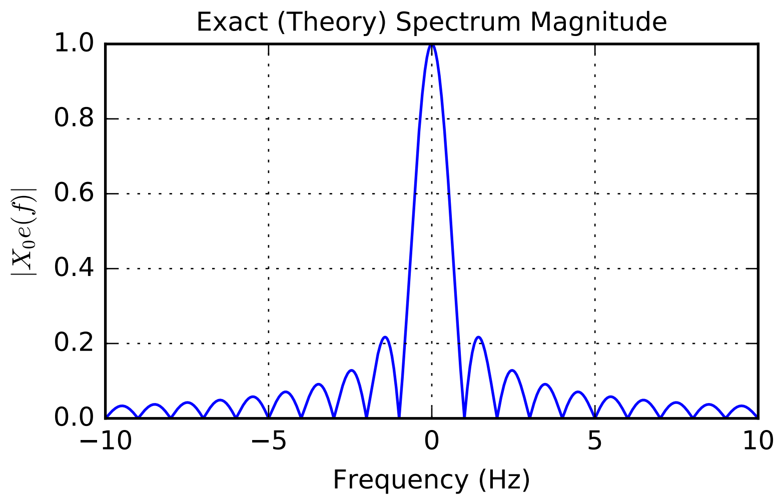
The output time axis starts at the sum of the starting values in x1 and x2 and ends at the sum of the two ending values in x1 and x2. The default extents of ('f','f') are used for signals that are active (have support) on or within n1 and n2 respectively. A right-sided signal such as $a^n * u[n]$ is semi-infinite, so it has extent 'r' and the convolution output will be truncated to display only the valid results.

Examples

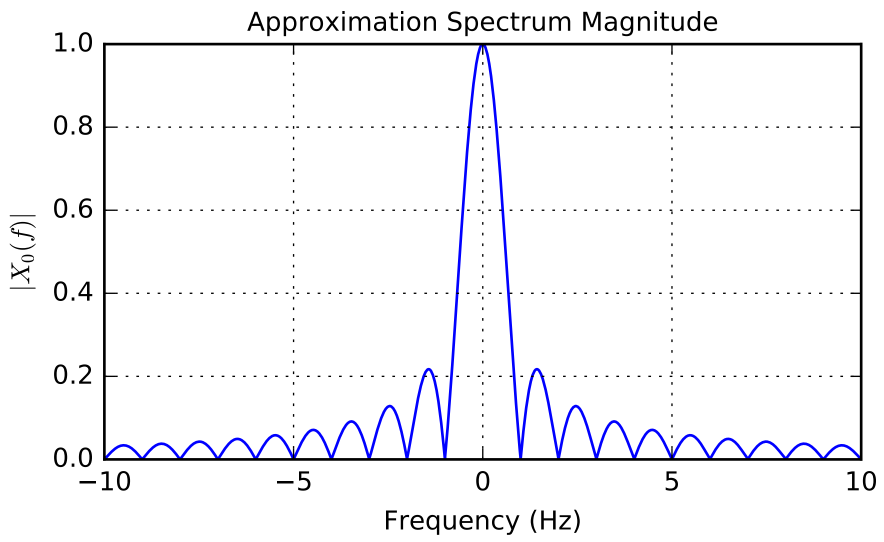
```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> import sk_dsp_comm.sigsys as ss
>>> fs = 100 # sampling rate in Hz
>>> tau = 1
>>> t = np.arange(-5,5,1/fs)
>>> x0 = ss.rect(t-.5,tau)
>>> plt.figure(figsize=(6,5))
>>> plt.plot(t,x0)
>>> plt.grid()
>>> plt.ylim([-0.1,1.1])
>>> plt.xlim([-2,2])
>>> 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
>>> 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 Fouier 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]

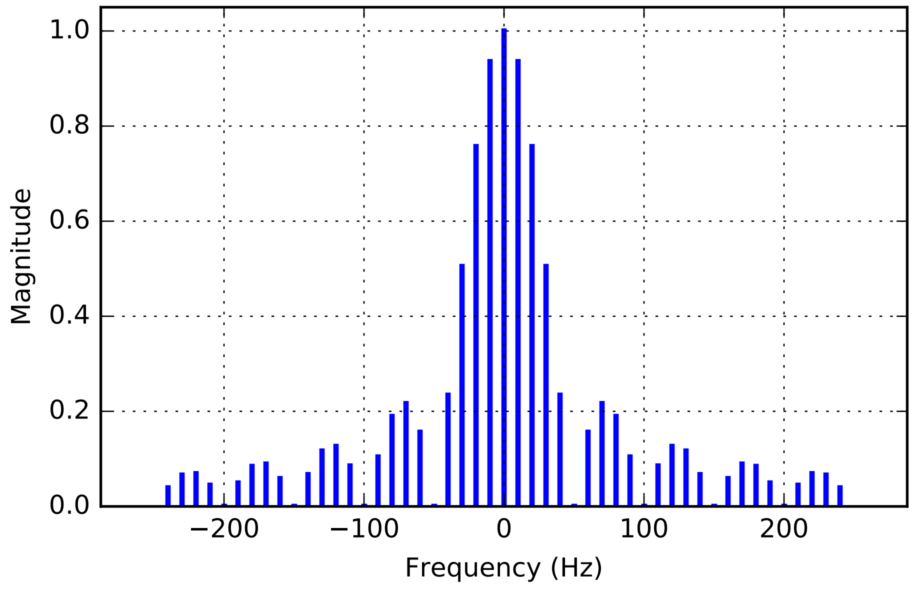
Notes

Since real signals are assumed the frequencies of fk are 0 and/or positive numbers. The supplied Fourier coefficients correspond.

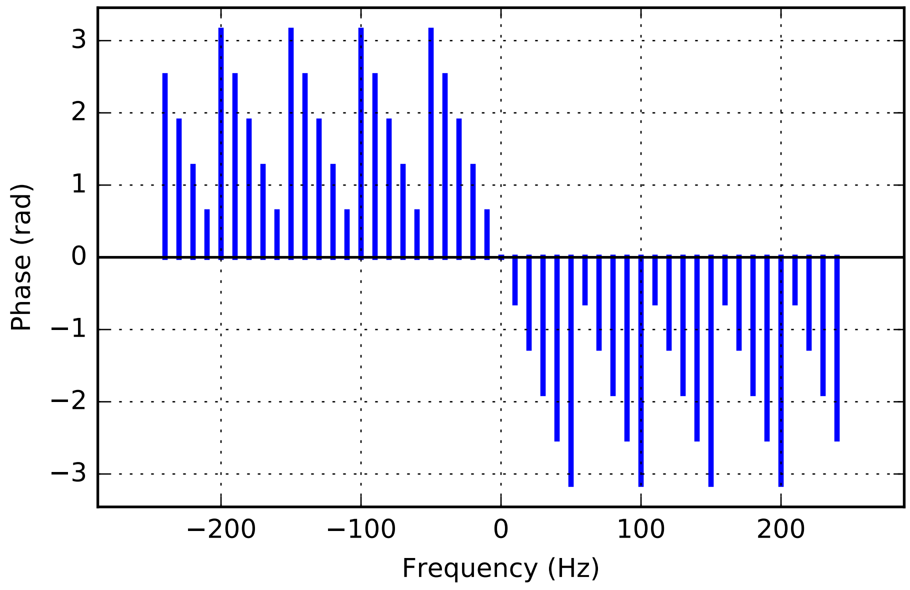
Examples

```
>>> 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)
```

```
>>> line_spectra(fk,Xk,'mag')
>>> plt.show()
```



```
>>> 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

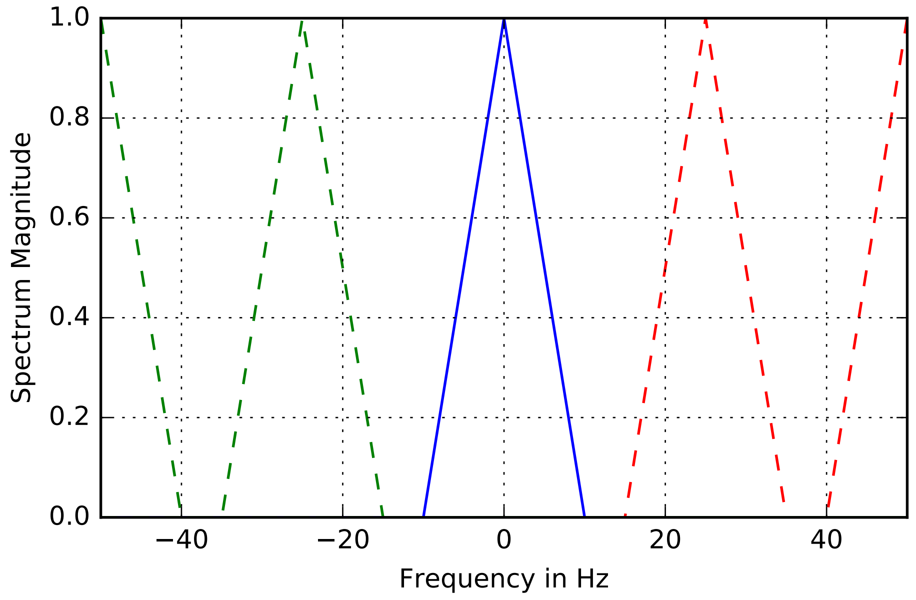
Nothing [A plot window opens containing the spectrum plot]

Examples

```
>>> import matplotlib.pyplot as plt
>>> from sk_dsp_comm.sigsys import lp_samp
```

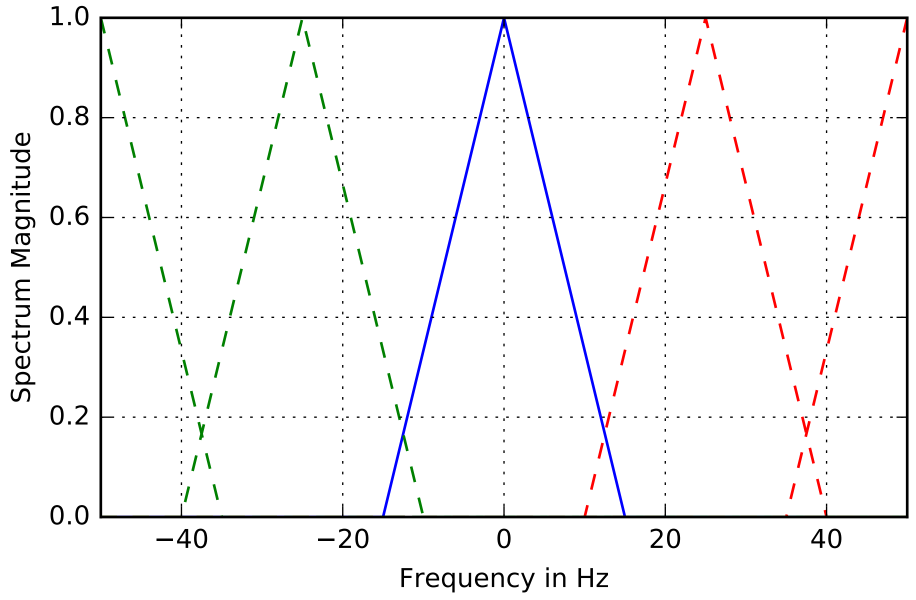
No aliasing as bandwidth 10 Hz < 25/2; fs > fb.

```
>>> 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]

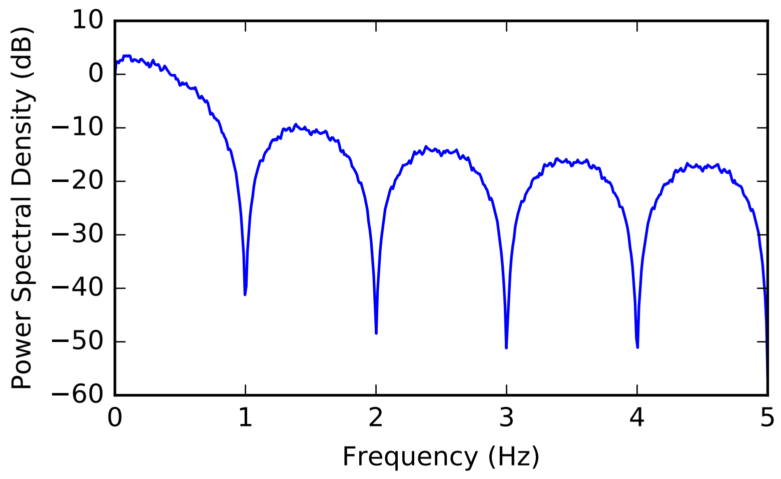
f [ndarray of frequency values]

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.`**`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]

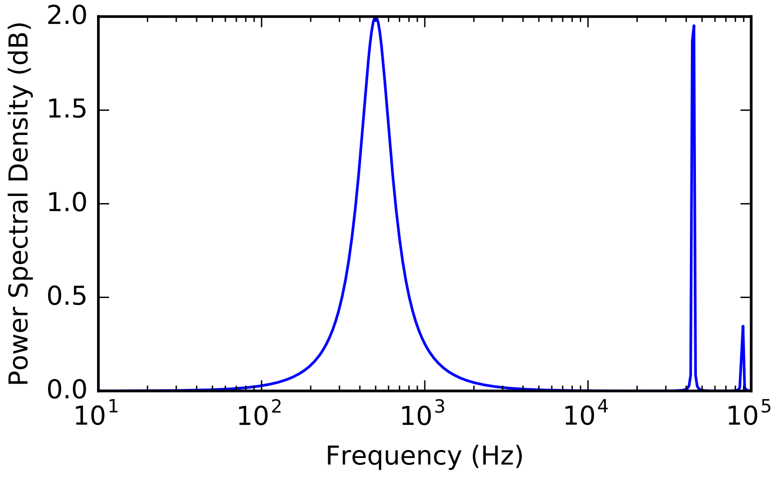
Returns

b [ndarray containing the numerator filter coefficients]

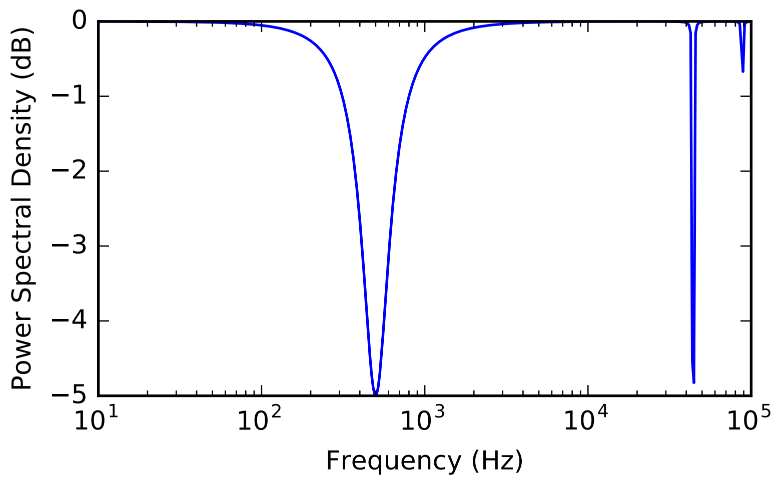
a [ndarray containing the denominator filter coefficients]

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.`**`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]

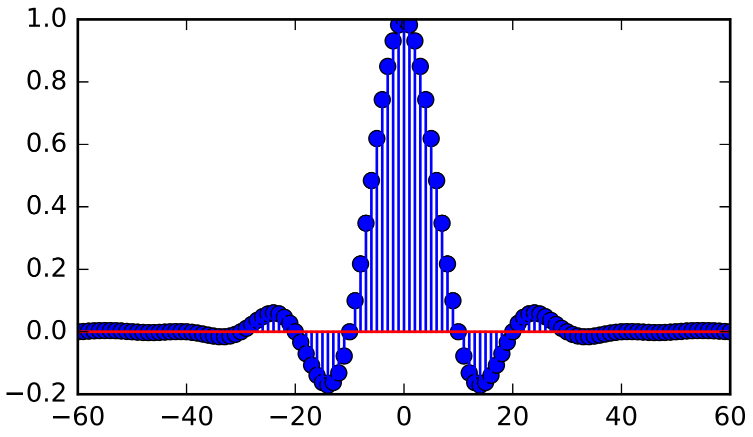
Notes

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]

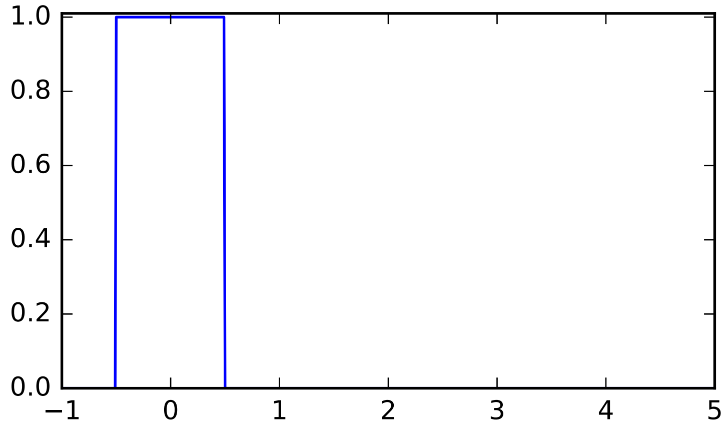
tau [the pulse width]

Returns

x [ndarray of the signal Pi(t/tau)]

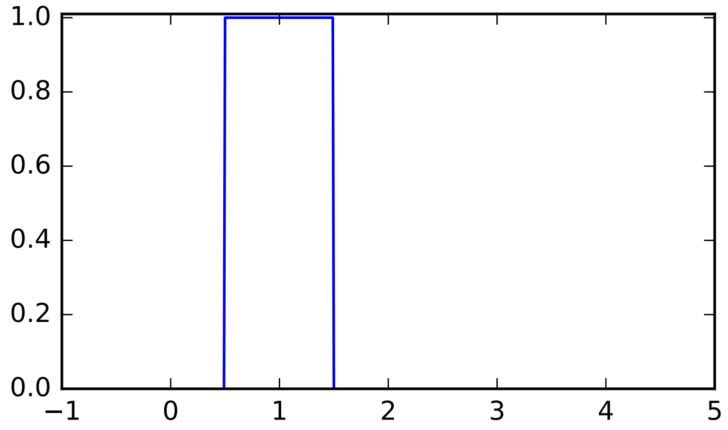
Examples

```
>>> 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]

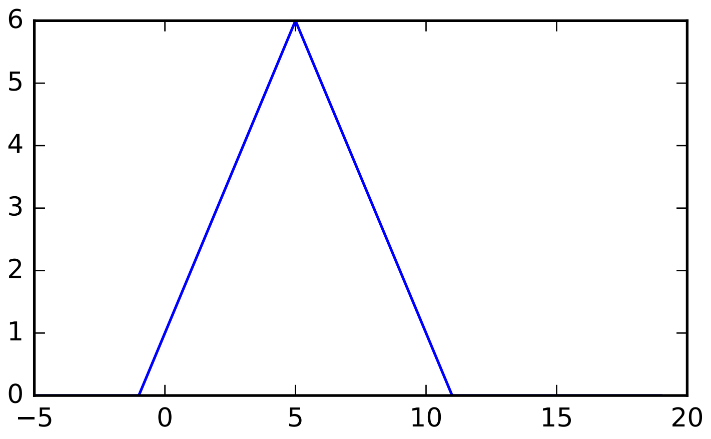
N_len [rectangle pulse duration]

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]

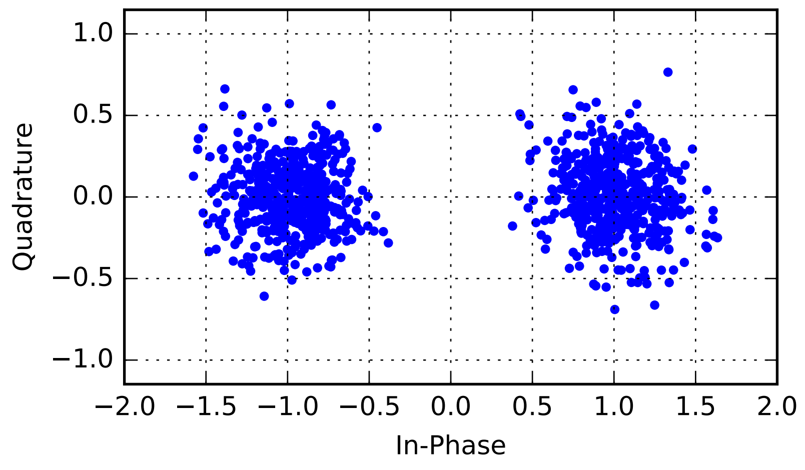
xQ [ndarray of the imaginary part of x following sampling]

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.`**`simpleQuant`**(*x*, *Btot*, *Xmax*, *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]

Btot [total number of bits in the quantizer, e.g. 16]

Xmax [quantizer full-scale dynamic range is [-Xmax, Xmax]]

Limit = Limiting of the form 'sat', 'over', 'none'

Returns

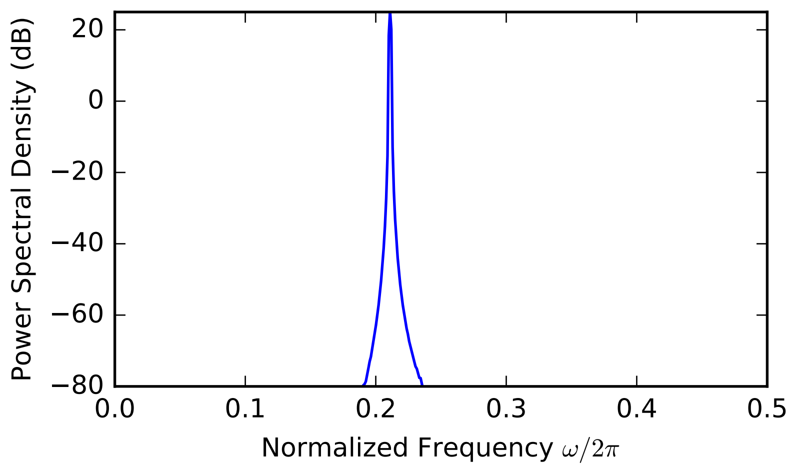
xq [quantized output ndarray]

Notes

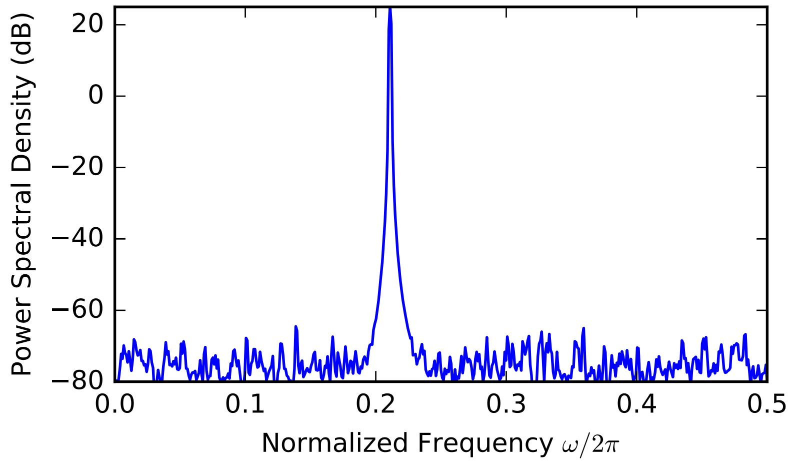
The quantization can be formed as e = xq - x

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.sinusoidAWGN(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.simpleQuant(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 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]]

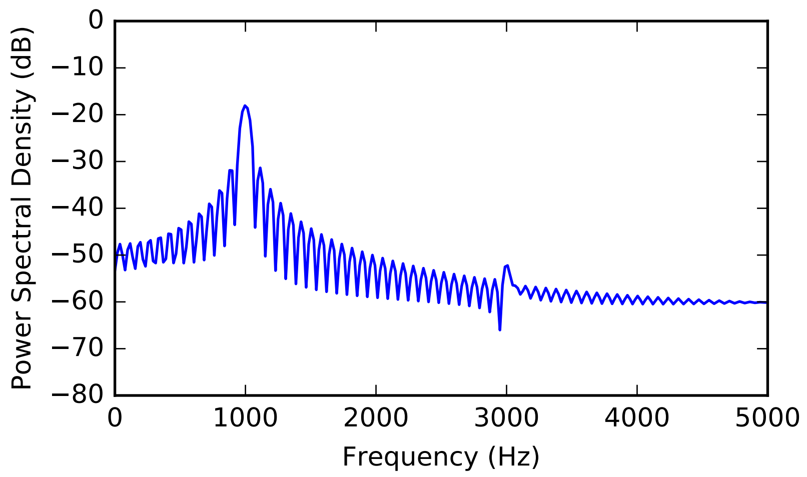
Sx [ndarray the power spectrum estimate]

Notes

The function also prints the maximum number of averages K possible for the input data record.

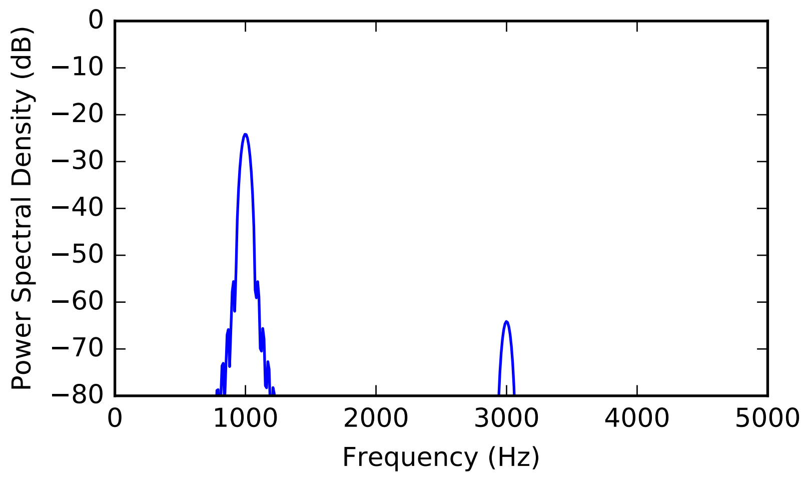
Examples

```
>>> 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.**sinusoidAWGN**(*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 = sinusoidAWGN(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

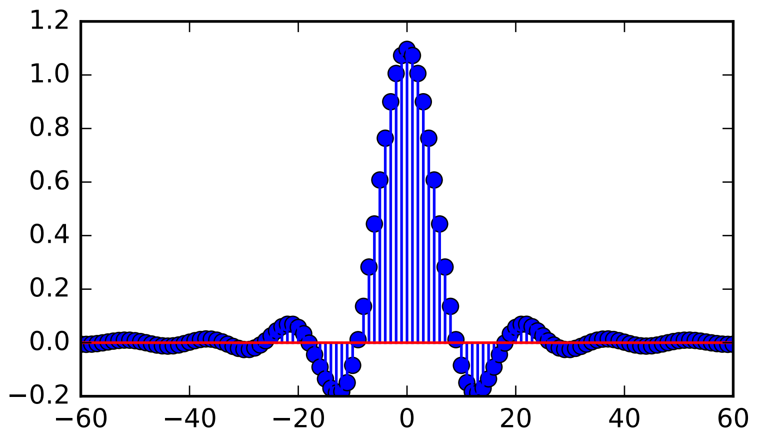
b [ndarray containing the pulse shape]

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

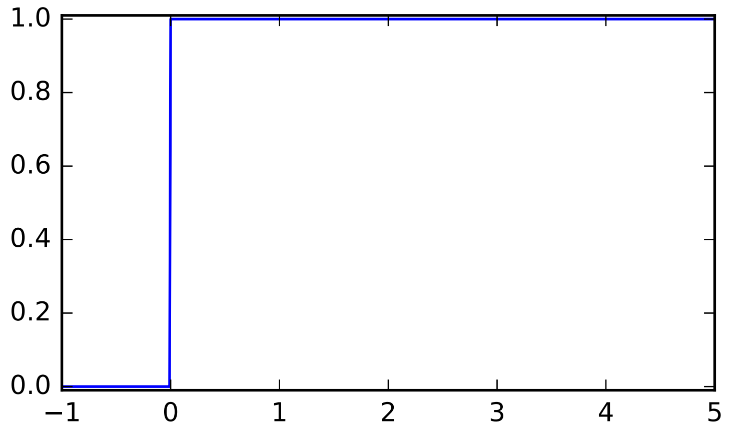
t [ndarray of the time axis]

Returns

x [ndarray of the step function signal u(t)]

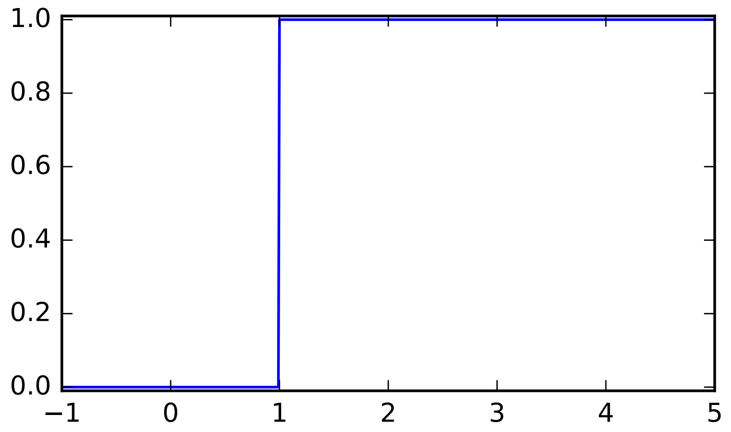
Examples

```
>>> 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]]

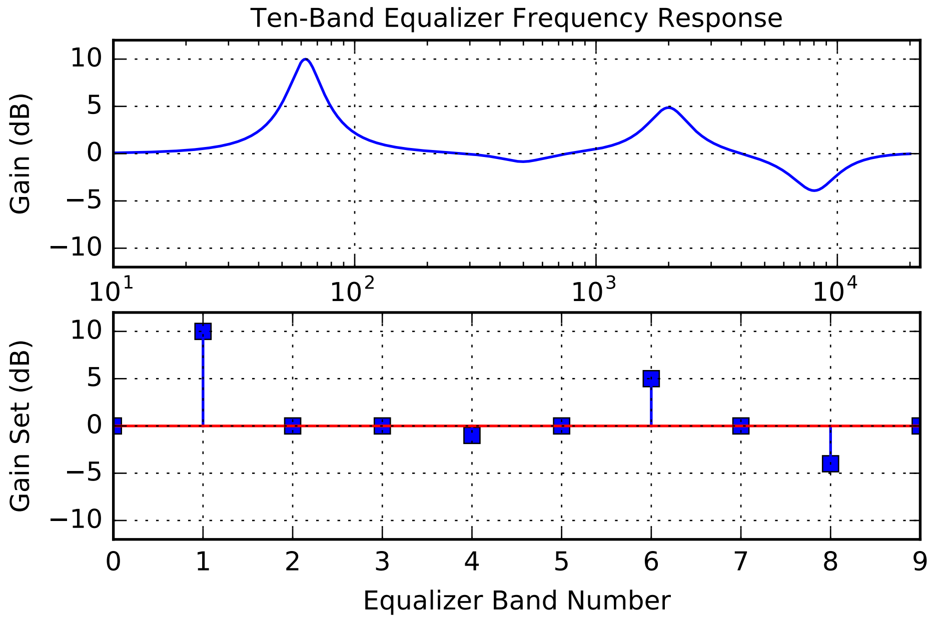
Q [Quality factor for each peaking filter (default 3.5)]

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]

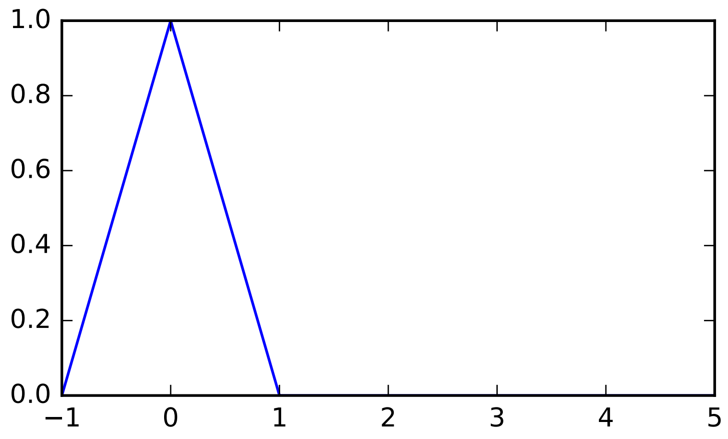
tau [one half the triangle base width]

Returns

x [ndarray of the signal Lambda(t/tau)]

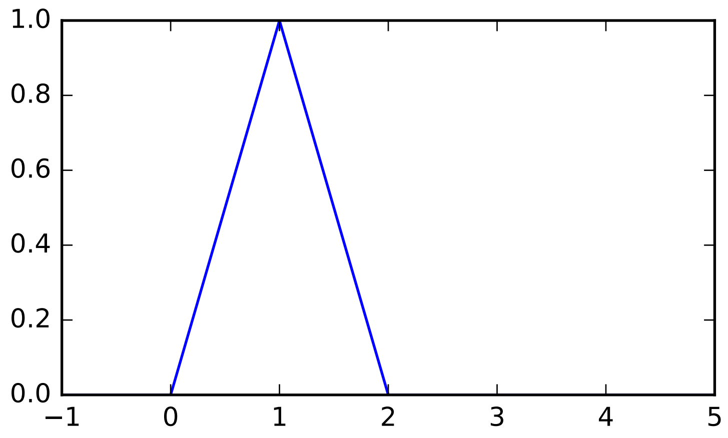
Examples

```
>>> 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)
```

1.11 synchronization

A Digital Communications Synchronization and PLLs Function Module

`sk_dsp_comm.synchronization.`**`DD_carrier_sync`**(*z*, *M*, *BnTs*, *zeta=0.707*, *type=0*)

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

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.`**`MPSK_bb`**(*N_symb*, *Ns*, *M*, *pulse='rect'*, *alpha=0.25*, *MM=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

```
>>> x,b,data = NRZ_bits(100,10)
>>> t = arange(len(x))
>>> plot(t,x)
```

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*)

[theta_hat, ev, phi] = PLL1(theta,fs,loop_type,Kv,fn,zeta,non_lin) Baseband Analog PLL Simulation Model ==

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

theta_hat = Output phase estimate of the input theta in radians

ev = VCO control voltage

phi = phase error = theta - theta_hat

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*)

[theta_hat, ev, phi] = PLL_cbb(theta,fs,loop_type,Kv,fn,zeta) Baseband Analog PLL Simulation Model ==

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

theta_hat = Output phase estimate of the input theta in radians

ev = VCO control voltage

phi = phase error = theta - theta_hat

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.`**`phase_step`**(*z*, *Ns*, *theta_step*, *Nsymb*)

Create a one sample per symbol signal containing a phase rotation step Nsymb into the waveform.

z = complex baseband signal after matched filter

Ns = number of sample per symbol

theta_step = size in radians of the phase step

Nstep = symbol sample location where the step turns on z_rot = the one sample symbol signal containing the phase step

Mark Wickert July 2014

`sk_dsp_comm.synchronization.`**`time_step`**`(`*z*, *Ns*, *time_step*, *Nstep*`)`

Create a one sample per symbol signal containing a phase rotation step Nsymb into the waveform.

z = complex baseband signal after matched filter

Ns = number of sample per symbol

time_step = in samples relative to Ns

Nstep = symbol sample location where the step turns on

z_step = the one sample per symbol signal containing the phase step

Mark Wickert July 2014

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index