A Discrete-Time Signal Processing Package for wxMaxima

August 11, 2025

Mark Wickert

Contents

1. Introduction

This package is mostly about doing numerical calculations and plotting for discrete-time signal processing analysis, modeling and simulation. I started to write in January 2025 as an experiment to see how far I could take things from my experience with other tools, most notable Python and Julia. I was pleasantly surprised! Maxima is not as fast as Python and Julia, but you get the power of a full *computer algebra system* (CAS). There are some interesting possibilities that I will explore in the examples. The package repository link.

My first exposure to Maxima, actually `wxmaxima` came in 2012-2013 when writing the Wiley Dummies series book, *Signals and Systems for Dummies* [1];. As a professor teaching signal processing and communications theory courses for many years, I had first started using `C/C++` and a plotting package as a modeling and simulation tool combination, I then moved to `MATLAB` and `Octave`. Along the way I also used `Mathematica` for symbolic, CAS type calculations. I stayed with MATLAB and Mathematica as a numeric/symbolic tool pair until I started doing work with a company where `Python` and the `scipy-stack` was being talked about as a replacement for MATLAB. I eventually spent some time using `sympy` in both Python and Julia Jupyter and Pluto notebooks.

Example in this and integrated the use of `sympy` into some of my teaching https://faculty.uccs.edu/mwickert/.

Some of the books that were used during the code writing include: M. Wickert *Signals and Systems for Dummies* [1], Oppenheim and Shafer *Discrete-Time Signal Processing* [2], Ziemer and Tranter *Principles of Communications* [3].

2. The Package Modules

In its present form the DSP package for wxmaxima consists of three modules: `DSPTools.mac`, `CommTools.mac`, and `FilterTools.mac`.

2.1. Loading Packages

To make it easy to `load()` the three code modules I suggest setting a path to the file location using the `push()` command to augment your existing `file_search_path`:

2.2. Maxima List Manipulation

As you with Maxima you find yourself wondering more and more about the available list manipulation functions. If you want to create a sublist from a list you can always use `makelist()` to create a new list from an existing list. Say you want to get at elements at the beginning, middle, or end of a list:

```
/* For Nstart and Nstop positive integers */
y: makelist(x[k],k,Nstart,Nstop)
```

Remember the first element of a list is at index 1 and the last element is at `length(x)`. Beyond the Maxima documentation itself, consider this great resource: *SysPlay's* Maxima list blog.

```
push("/Users/myfolder/myfunctions/maxima/###.mac", file_search_maxima)$

load("DSPTools.mac")$
load("CommTools.mac")$
load("FilterTools.mac")$
```

2.3. Plotting with Exact Expressions

Depending upon the length of a list containing exact expressions there may be issues. When in doubt convert the list to a numeric list first using the `float()` function:

```
y_f:float(y)
```

2.4. `DSPTools.mac`

Table 1: General discrete-time signal processing functions contained in the module `DSPTools.mac`; parameters in brackets are optional.

Function name	**Purpose/Description**
x:ones(*N*)	Generate a list, x, of N ones.
x:zeros(*N*)	Generate a list, x, of N zeros.
x:dstep(*n*,*n*_shift)	Generate a unit step sequence [2] list, $x[n] = u[n - n_{\text{shift}}]$, where n is a list of indices covering n_1 to n_2. The variable n_{shift} shifts the starting point along n.
x:drect(*n*,*n*_shift,*M*__len)	Generate a rectanglular pulse $x[n] = \text{rect}[n - n_{\text{shift}}]$ of M_len samples duration where n is a list of indices covering n_1 to n_2.
stemlist(y_data,[x_label],[y_label],[title1],[ylim], first_x_value)	A stem plot of the list y_data with optional x-axis, y-axis, and title labels. The optional variable ylim is a two element list to set the y-axis limits. The required variable first_x_value sets the index of the first element plotted, i.e., effectively defines the location of the entire list along the time axis.
plotlist(y_data,[x_label],[y_label],[title1],[ylim], first_x_value)	Plot a list by connecting the list points, otherwise this function has the same options as stemlist().
z:listconv(x,y)	Convolve [1] the sample values of the N_x point list x with the sample values of the N_y point list y. The returned sequence z has length $N_x + N_y$.
y:fir_filt(b_coef,x)	Finite impulse response (FIR) filter [1], [2] the list x with the coefficients list b_k of length M. In the z-domain the system function is a Mth order polynomial over a Nth-order denominator. The function returns y, a list of the same length as x. This implements the feed-forward difference equation of the classical linear constant coefficient difference equation (LCCDE) for $n \geq 0$: $$y[n] = \sum_{k=0}^{M-1} b_k x[n-k] \quad (1)$$
y:filt(b_coef,a_coef,x,[y_ic],[x_ic)]	Infinite impulse response (IIR) filter [1], [2] the list x with the feed forward coefficients list b_k of

Function name	Purpose/Description
	length $M+1$ and the feedback coefficient list a_k of length N (coefficient a_0 is assumed to be unity). The function returns y, a list of the same length as x. This implements a LCCDE feed-forward/back difference equation for $n \geq 0$: $$y[n] = \sum_{k=0}^{M} b_n x[n-k] - \sum_{k=1}^{N} a_k y[n-k] \quad (2)$$ Non-zero initial conditions can be set via optional lists y_ic and x_ic. You can thus set nonzero values for $y[-1], ..., y[-N]$ and $x[-1], ..., x[-M]$.
freqz(b_coef,a_coef,[xlim],[ylim],[mode],[title1],[f_axis],[fs])	Calculate and plot the Fourier transform, magnitude and phase, of a rational system function on the unit circle in the z-plane, i.e. $$H(e^{j\omega}) = \frac{\sum_{k=0}^{M} b_k e^{j\omega}}{\sum_{k=0}^{N} a_k e^{j\omega}}, 0 \leq \omega < 2\pi \quad (3)$$ or using substitution for real frequency f in Hz in place of ω, let $\omega \rightarrow 2\pi f/f_s$ $$H\left(e^{j2\pi f/f_s}\right) = \frac{\sum_{k=0}^{M} b_k e^{j2\pi f/f_s}}{\sum_{k=0}^{N} a_k e^{j2\pi f/f_s}} \quad (4)$$ where ω is the Fourier frequency variable in radians/samples and f_s is the sampling rate in Hz, making f the Fourier frequency variable in Hz. The list b_coef is of length $M+1$ and the list a_coef is of length $N+1$. The optional two element list xlim lets you zoom the frequency axis eith in radians or Hz depending on f_axis being “Hz”, the default, or “rad/samp”. The optional variable mode is the string “ampl_dB” (default) or “phase”. The optional two element list ylim lets you zoom the amplitude or phase axis. The optional variable fs (default 1) sets the sampling rate when f_axis is “Hz”. To Fourier transform a signal load the values into b_coef and load [1] into a_coef. If complex values are present for either coefficient set values use xlim to cover say [-%pi, %pi] in rad/sample or [-fs/2,fs/2] in Hz.

Function name	**Purpose/Description**
zplane(b_coef,a_coef,[ri_minmax])	Calculate and plot the poles and zeros of the rational system function having coefficients b_coef (num) and a_coef (den). The optional four element list ri_minmax = [minr, maxr, mini, maxi] allows you to zoom to specific rectangular region of the z-plane. Currently repeated roots are not indicated.
y:downsample(x,M_dn,[p_phase])	The list y is a downsampled [2] version of the list x, with the optional variable p_phase setting starting point, $0 \leq p_{\text{phase}} \leq M_{\text{dn}} - 1$, i.e., $$y[n] = x\left[\left(n - p_{\text{phase}}\right) M_{\text{dn}}\right] \qquad (5)$$ To form a decimator band limit the spectrum of $x[n]$ to $\lvert\omega\rvert \leq \pi / M_{\text{dn}}$.
y:upsample(x,L_up)	The list y is an upsampled [2] or expanded version of the list x, which means $L_{\text{up}} - 1$ zero samples are stuffed in between each sample of x, i.e., $$y[n] = \sum_{k=\infty}^{\infty} x[k] \delta\left[n - k/L_{\text{up}}\right], \qquad (6)$$ where the discrete sample function is defined as $\delta[n] = 1$ for $n = 0$ and $\delta[n] = 0$ for $n \neq 0$.
X:DFT_pt(x,k)	The discrete Fourier transform (as defined in electrical engineering texts) of the N point list x at the single point k i.e., $$X_k = \sum_{k=0}^{N-1} x[n] e^{-j2\pi kn/N}, k \in Z \qquad (7)$$ Note this a mod N function so k and $k + N$ produce the same result.
X:DFT(x)	The discrete Fourier transform of N point list x as an N-point list of frequency domain points over $0 \leq k \leq N - 1$, i.e., $$X[k] = \sum_{k=0}^{N-1} x[n] e^{-j2\pi kn/N}, 0 \leq k < N \quad (8)$$ In the code the list index runs from 1 to N. The Maxima functions `cabs()`, `carg()`, `polarform()`, `rectform()`, `realpart()`, and `imagpart()` may be useful and simplification try `gfactor()`.

Function name	**Purpose/Description**
x:IDFT_pt(X,n)	The discrete inverse Fourier transform (as defined in electrical engineering texts [1], [2]) of the N point frequency domain list X at the single point n i.e., $$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X[n]e^{j2\pi nk/N}, n \in Z \qquad (9)$$ Note this a mod N function so n and $n+N$ produce the same result.
x:IDFT(X)	The discrete inverse Fourier transform of N-point list X as an N point list of time (sequence) domain points over $0 \leq n \leq N-1$, i.e., $$x[k] = \sum_{k=0}^{N-1} x[n]e^{-j2\pi kn/N}, 0 \leq k < N \quad (10)$$ In the code the list index runs from 1 to N.
fft_dsp(x,[Np2])	Numerically compute the discrete Fourier transform of list x (as defined in electrical engineering texts [1], [2]). The package uses `load("fft")`, which means a power of two is required for the returned frequency domain list. The optional variable Np2, chosen as a power two, can be used to truncate or zero pad x.
ifft_dsp(X)	Numerically compute the discrete inverse Fourier transform of list X (as defined in electrical engineering texts [1], [2]). The package uses `load("fft")`, which means a power of two is required for the input frequency domain list.
y:log10(x,[float_mode)]	Returns log() base 10, where the optional variable `float_mode` is by default `true` to give a numerical value. For dB values 10 or $20 \cdot \log10(x, [])$.
y:sinc(x)	Returns the sinc() function defined as $\sin(\pi x)/(\pi x)$.
w:hann(L)	Returns an L-point list w which is the well known Hann window function [2] used in filter design and non-parametric spectral estimation, i.e., $$w[n] = \frac{1}{2}\left[1 - \cos\left(2\pi\frac{n}{L-1}\right)\right], \qquad (11)$$ for $0 \leq n \leq L-1$. See below `psd()` function.

Function name	**Purpose/Description**
psd(x,N_fft,fs,[ylim],[x_label],[y_label],[title1],[real_signal],[overlap_percent],[scale_noise)]	Implements Welch's method of averaged periodogram spectral estimation [2]. From the input list L overlapping using `hann()` windowed overlapping N-point sub-records formed from the input list x. Beyond the optional plot label arguments, the optional variable overlap_percent sets the percent overlap between sub-records. The default is 50. The optional boolean variable scale_noise sets the dB amplitude level to proper noise spectral density for white noise (default `true`) or approximate dB amplitude levels for sinusoidal signals (`false`). To explain further, when `true` noise like signals will display as dB amplitude values that reflect $10 \log_{10}$ the noise power or variance of the spectrum and when `false` the dB amplitude of a complex or real sinusoid has amplitude A, the display value is $10 \log_{10}$ of A^2 or $A^2/2$, respectively.
plotlist2(y_data1,y_data2,[x_label],[y_label],[title1],[ylim], first_x_value)	Similar to plotlist() except now there are two input data lists, y_data1 and y_data2, to be plotted. The required first_x_value is a two element list allowing arbitrary offsets between the two data sets.
plotlist3(y_data1,y_data2,y_data3,[x_label],[y_label],[title1],[ylim],first_x_value)	Similar to plotlist() except now there are three input data lists, y_data1, y_data2, and y_data3, to be plotted. The required first_x_value is a three element list allowing arbitrary offsets between the three data sets.
plotlist4(y_data1,y_data2,y_data3,y_data4,[x_label],[y_label],[title1],[ylim],first_x_value)	Similar to plotlist() except now there are four input data lists, y_data1, y_data2, ydata3, and y_data4, to be plotted. The required first_x_value is a four element list allowing arbitrary offsets between the four data sets.

2.5. `CommTools.mac`

Table 2: Digital communications signal processing functions contained in the module `CommTools.mac`; parameters in brackets are optional.

Function name	**Purpose/Description**
x_bb:NRZ_bits(N_bits,Ns,[pulse_shape],[pulse_alpha],[M])	Generates a non-return-to-zero binary communications waveform in the real list x_bb. The waveform can be viewed as a baseband band (bb) form of binary phase-shift keying (BPSK) in the discrete-time domain. The variable N_bits sets the

Function name	**Purpose/Description**
	number of bits to generate using a random *coin-toss* sequence of data bits. The variable Ns sets the number of waveform samples per data bit. There are three optional parameters. The first is `pulse_shape` which can be "rect" (default), "src" for square-root raised cosine, or "rc" for raised cosine. The variable alpha sets the *excess bandwidth* parameter used in the "src" and "rc" pulse shapes. The variable alpha (α, default is 0.25) sets the excess spectral bandwidth parameter of the "src" and "rc" pulse shapes. The variable alpha controls the spectral occupancy of pulse shaped digital modulation such as found in cellular WiFi wireless schemes. Finally variable M, for the case of "src" or "rc" pulse, sets the half width of the pulse duration as $M \cdot N_s$ (the default is $M = 6$).
r_bb:cpx_awgn(x,Es2N0_dB,Ns)	This function adds complex white Gaussian noise to a communications waveform such as that generated by `NRZ_bits`. The fact that complex Gaussian random variates are added to a real signal is to create a received signal model of a communications waveform at the front end of a receiver impacted by thermal noise in the analog receiver electronics. The input list x is assumed to be real the returned list r_bb is a complex list. The variable `Es2N0_dB` sets the variance of the Gaussian (normal) variates to correspond to a desired received energy-per-symbol to noise power spectral density, $(E_s/N_0)_{\text{dB}}$. Here the symbols are bits, but we could form a quadriphase baseband waveform in which case we would have two bits per symbol. The third variable Ns tells the function how many samples per symbol in the waveform.
eye_plot(x_bits,span,[trim_head])	This function is used to plot a stack-up or overlay of digital communications waveform slices. The waveform, `x_lim`, is assumed to be real. The variable `span` is typical set to display two bit periods. The optional variable `trim_head` defaults to zero but is typically set transient in the input waveform when pulse shaping is present. A good choice is $12 \times N_s$ when using the default pulse shape variable $M = 6$.

Function name	**Purpose/Description**
constellation(z,Ns,[N_timing],[trim_head],[xy_lim])	This function plots samples of the full complex baseband waveform in the list `z` at integer multiples of the symbol period, N_s, with `N_timing` the sampling instant within each period. The `trim_head` variable behaves the same as it does in `eye_plot`. The optional variable `xy_limit` sets the plot dimensions (equal axis limits) with the default 1.25.
sinc(x)	The familiar $\sin(\pi x)/(\pi x)$ function as found in signals and systems. The limit at $x = 0$ is set to unity.
h:sqrt_rc_imp(Ns,alpha,[M],[eps_scale])	This function generates a finite duration impulse response for a square-root raised cosine pulse shape as found in pulse shaped digital communications. The variable `Ns` sets the number of samples per symbol in generating the impulse response. The function `NRZ_bits` uses `h`, to its waveform, but `h` is also to generate the filter coefficients in a matched filter receiver when the transmit waveform also uses `h`. The variable `alpha` is the excess bandwidth factor, typically between 0.5 and 0.25. The optional variable `M` sets the number of `Ns` sample periods left and right impulse response tails. There is also a sample in the center, so the total filter length is $2 \times M \times N_s + 1$. The optional variable `eps_scale` (default is one) controls the switching point in this piecewise function. Changing it helps avoid discontinuities in the pulse shape.
h:rc_imp(Ns,alpha,[M])	This function generates a finite duration impulse response for a raised cosine pulse shape as found in pulse shaped digital communications. The variable `Ns` sets the number of samples per symbol in generating the impulse response. The function `NRZ_bits` uses `h`, to its waveform, but `h` is also to generate the filter coefficients in a matched filter receiver when the transmit waveform also uses `h`. The variable `alpha` is the excess bandwidth factor, typically between 0.5 and 0.25. The optional variable `M` sets the number of `Ns` sample periods left and right impulse response tails. There is also a sample in the center, so the total filter length is $2 \times$

Function name	Purpose/Description
	$M \times N_s + 1$. The square-root raised cosine (SRC or RRC) and raised cosine (RC) pulses are related. Ideally the RC is the convolution of the SRC with itself.

2.6. `FilterTools.mac`

Table 3: Filter design related signal processing functions contained in the module `FilterTools.mac`; parameters in brackets are optional.

Function name	Purpose/Description
w:w_kaiser(As_dB,M)	Returns a real list, `w`, corresponding to a Kaiser window function used in linear phase FIR filter design. The variable As_dB sets the stopband attenuation in dB of the filter. The length of the filter is the integer variable M. The coefficients are used behind the scenes in the four firwin filter design function below this table. Note `FilterTools.mac` requires `load("bessel")` to be successful
b:firwin_kaiser_lpf(f_pass, f_stop, d_stop, fs, [n_bump)]	This function returns a list that contains FIR filter coefficients for a linear phase lowpass filter using the Kaiser window function [2]. The variables f_pass and f_stop (clearly $f_{\text{pass}} < f_{\text{stop}}$) set the desired passband and stopband critical frequencies respectively, in Hz relative to the sampling frequency variable fs. The variable d_stop sets the stopband attenuation in dB. The passband ripple is related to the stopband attenuation in a classical windowed FIR design. The optional variable n_bump is an integer used to step the filter length (filter order + 1) up or down by steps as small as ± 1 to fine tune the design. The coefficients are used in the LCCDE functions `fir_filt` and `filt`.
b:firwin_kaiser_hpf(f_stop, f_pass, d_stop, fs, [n_bump])	This function returns a list that contains FIR filter coefficients for a linear highpass filter using the Kaiser window function [2]. The variables f_stop and f_pass (clearly $f_{\text{stop}} < f_{\text{pass}}$) set the desired stopband and passband critical frequencies respectively, in Hz relative to the sampling frequency variable fs. The variable d_stop sets the stopband attenuation in dB. The passband ripple is related to the stopband attenuation in a classical windowed FIR design. The optional variable n_bump is an integer used to step the filter length (filter order +

Function name	**Purpose/Description**
	1) up or down by steps as small as ±1 to fine tune the design. The coefficients are used in the LCCDE functions `fir_filt` and `filt`.
b:firwin_kaiser_bpf(f_stop1, f_pass1, f_pass2, f_stop2, d_stop, fs, [n_bump])	This function returns a list that contains FIR filter coefficients for a linear phase bandpass filter using the Kaiser window function [2]. The variables f_stop1, f_pass1, f_pass2, and f_stop2 (clearly $f_{\text{stop1}} < f_{\text{pass1}} < f_{\text{pass2}} < f_{\text{stop2}}$) set the desired lower stopband edge, lower passband edge, upper passband edge, and upper stopband edge critical frequencies respectively, in Hz relative to the sampling frequency variable fs. The variable d_stop sets the stopband attenuation in dB. The passband ripple is related to the stopband attenuation in a classical windowed FIR design. The optional variable n_bump is an integer used to step the filter length (filter order + 1) up or down by steps as small as ±1 to fine tune the design. The coefficients are used in the LCCDE functions `fir_filt` and `filt`.
b:firwin_kaiser_bsf(f_stop1, f_pass1, f_pass2, f_stop2, d_stop, fs, [n_bump])	This function returns a list that contains FIR filter coefficients for a linear phase bandstop filter using the Kaiser window function [2]. The variables f_pass1, f_stop1, f_stop2, and f_pass2 (clearly $f_{\text{pass1}} < f_{\text{stop1}} < f_{\text{stop2}} < f_{\text{pass2}}$) set the desired lower passband edge, lower stopband edge, upper stopband edge, and upper passband edge critical frequencies respectively, in Hz relative to the sampling frequency variable fs. The variable d_stop sets the stopband attenuation in dB. The passband ripple is related to the stopband attenuation in a classical windowed FIR design. The optional variable n_bump is an integer used to step the filter length (filter order + 1) up or down by steps as small as ±1 to fine tune the design. The coefficients are used in the LCCDE functions `fir_filt` and `filt`.
ba:fir_iir_notch(fi,fs,r)	This function returns a six element list, ba, which corresponds to a single section biquadratic filter of the form

Function name	Purpose/Description
	$$H(z) = \frac{1 - 2\cos\left(2\pi \frac{f_i}{f_s}\right) z^{-1} + z^{-2}}{1 - 2r\cos\left(2\pi \frac{f_i}{f_s}\right) z^{-1} + r^2 z^{-2}} \quad (12)$$ that forms a notch filter. The first three elements are the numerator coefficients (b_0, b_1, b_1) and the last three elements are the denominator coefficients (a_0, a_1, a_2) unless $r = 0$, in which case the filter becomes FIR with a pair of conjugate zeros on the unit circle at the notch frequency $2\pi f_i / f_s$ The variable fi specifies the location of the notch frequency relative to the sampling rate variable fs. The third variable is r ($0 \leq r < 1$), which control the pole radius of conjugate poles just inside the unit circle from the zeros. To parse `ba` for use `rest(ba,-3)` for `b` and `rest(ba,3)` for `a`

3. DSP Tools Examples

In this section we provide some examples on the use of the functions found in the DSP Tools code module. The examples were run on `macOS` with the wxmaxima installation managed using Homebrew. To kick things off once the wxmaxima GUI is running I first entered the commands

```
(%i1) push("/Users/markwickert/Documents/Projects/maxima/###.mac",
      file_search_maxima)$
(%i2) load("DSPTools.mac")$
(%i3) load("CommTools.mac")$
(%i4) load("FilterTools.mac")$
```

In my setup I have the three `*.mac` files located in the directory `maxima`.

3.1. Using the Primitives `zeros()` and `ones()` with the Maxima `append()` Function

Here we construct a simple test signal composed of groupings of zeros and ones samples. we first create a subsequence of five ones in variable `x1`, 10 zeros in variable `x2` and five zeros in variable `x3`. We then append the subsequences into one lone sequence using the Maxima list function `append()`.

Using the Primitives zeros() and ones() from `DSPTools.mac` and the Maxima append() Function

```
(%i14)  x1:ones(5);
(x1)  [1,1,1,1,1]
(%i15)  x2:zeros(10);
(x2)  [0,0,0,0,0,0,0,0,0,0]
(%i16)  x3:zeros(5);
(x3)  [0,0,0,0,0]
(%i18)  y:append(x3,x1,x2);
(y) [0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0]
stemlist(y,[],[],[],[0,1.2],-5);
```

$$y[n] = [0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0], 0 \leq n \leq 5+5+10-1 = 19 \qquad (13)$$

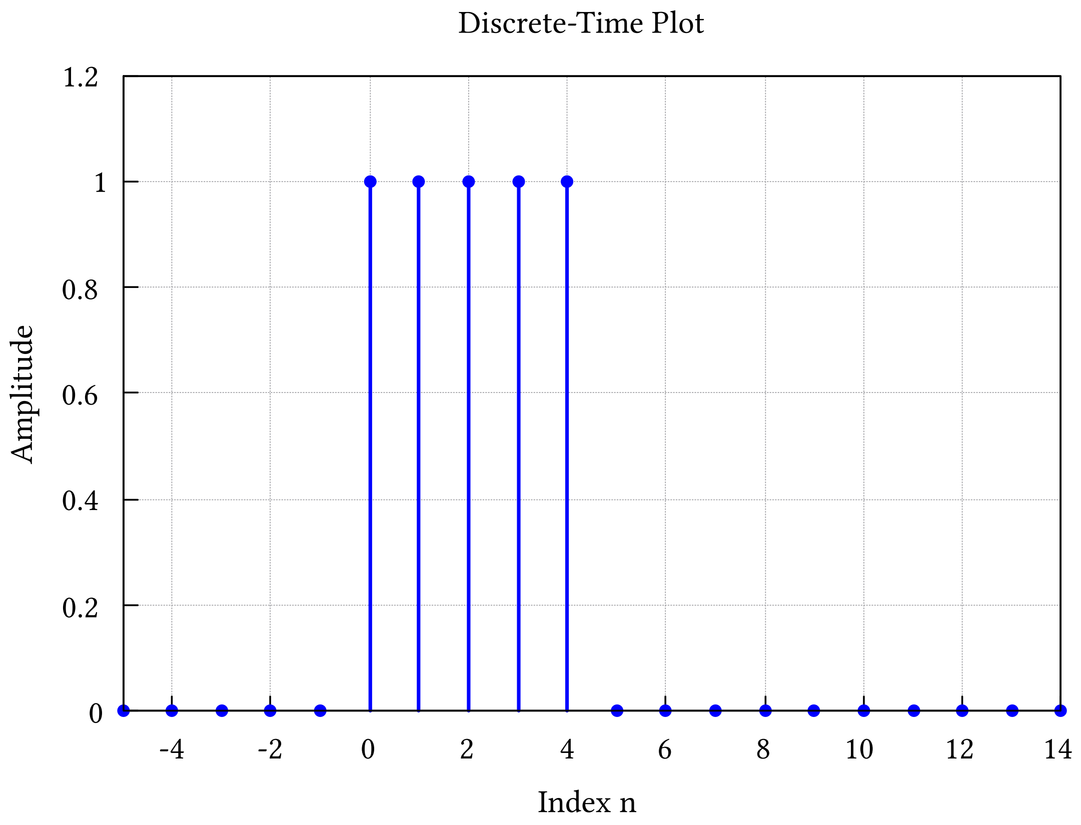

Figure 1: Appending primitive sequences and plotting as a `stem plot`.

3.2. Using `makelist()` and `if, then, elseif` and `else` to Create a Piecewise Sequence

Here we use the list manipulation constructs of the Maxima language to directly create a piecewise sequence.

A Piecewise Continuous Sequence using makelist()

```
(%i38)  y2:makelist(if n < 0 then 0 elseif n < 8 then n+1/20*n^2 else 0,n,-5,14);
(y2)  [0,0,0,0,0,0,21/20,11/5,69/20,24/5,25/4,39/5,189/20,0,0,0,0,0,0,0]],
[0,10],-5);
(%i139) stemlist(y3,[],[],"Piecewise Sequence",[],-5);
```

$$y_2[n] = \begin{cases} 0, & n < 0 \\ n + \frac{n^2}{20}, & 0 \le n < 8 \\ 0, & n \ge 8 \end{cases} \tag{14}$$

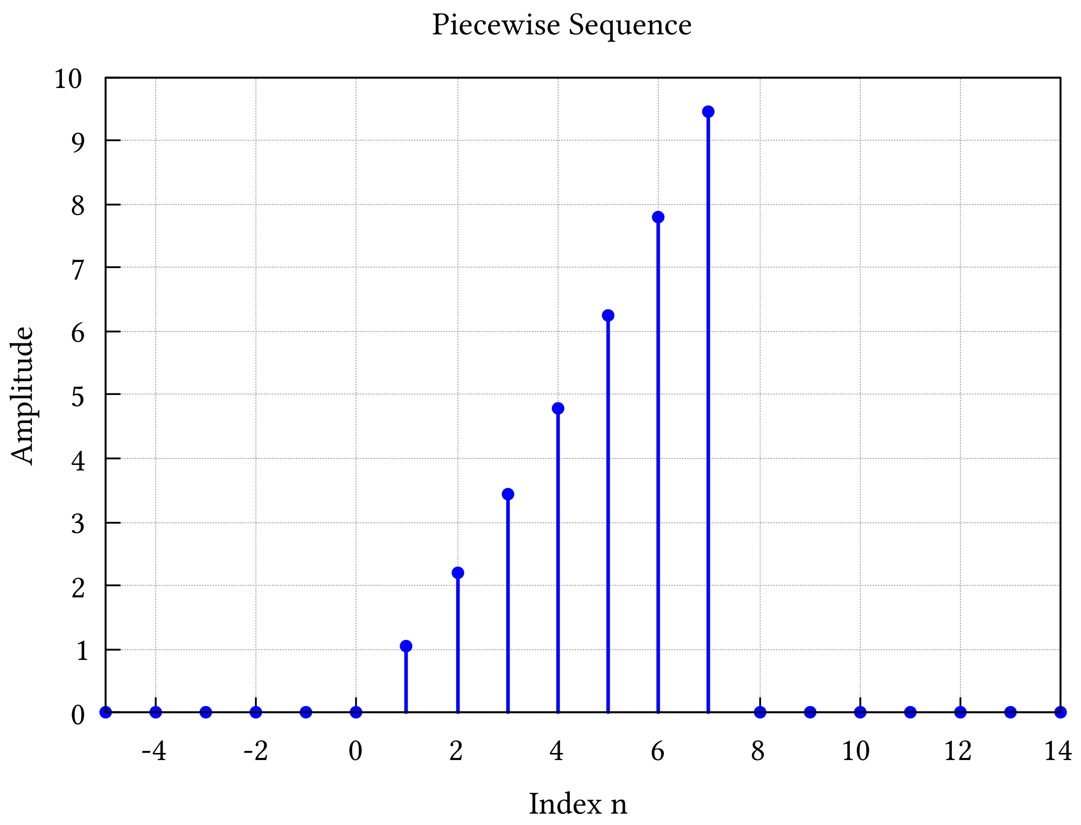

Figure 2: Piecewise function `stem plot`.

3.3. Explore Causal Constant Coefficient Difference Equations using `filt()`

DSP has traditionally used linear constant coefficient difference equations (LCCDEs) as a means to modify discrete-time signals. The CAS capabilities of Maxima allows the `filt()` function to explore both symbolic and numerical solutions of causal difference equations with both zero and non-zero initial conditions. The general form of the causal LCCDE is

$$y[n] = ay[n-1] + x[n], n \geq= 0 \tag{15}$$

where in this example $a = 3/5$ and since we have one feedback term the possible nonzero initial condition $y[-1]$.

Beyond the time domain relation responses we also consider the system frequency response, which is the Fourier transform of the impulse response. The frequency response is a complex function of the radian frequency ω or when considering $h[n]$ samples taken from the continuous-time t at spacing $1/f_s$, f_s being the sampling rate, means that $\omega = 2\pi f/f_s$. The function `freqz()` the calculations and produces plots in ω or f in Hz. By default $f_s = 1$.

From the z-transform of the impulse response obtained or by z-transforming both sides of the LCCDE we obtain the *system function* and plots its pole and zeros using `zplane()`

Using the LCCDE function `filt` compute outputs to various inputs with initial conditions

```
Impulse response
(%i146) y1:filt([1],[1,-6/10],[1,0,0,0,0,0,0,0,0,0,0],[0],[]);
(y1)  [1,3/5,9/25,27/125,81/625,243/3125,729/15625,2187/78125,6561/390625,
19683/1953125,59049/9765625]
(%i147) stemlist(y1,[],[],"1st-Order LCCDE Impulse Response",[],0);
Delayed Input Impulse with Non-zero Initial Conditions
(%i148) y2:filt([1],[1,-6/10],[0,0,1,0,0,0,0,0,0,0,0],[-1],[]);
(y2)  [-(3/5),-(9/25),98/125,294/625,882/3125,2646/15625,7938/78125,23814/390625,
71442/1953125,214326/9765625,642978/48828125]
A symbolic Solution
(%i149) y3:filt([1],[1,-a],[0,0,b,0,0,0,0,0,0,0,0],[-c],[]);
(y3)  [-(a*c),-(a^2*c),b-a^3*c,a*(b-a^3*c),a^2*(b-a^3*c),
a^3*(b-a^3*c),a^4*(b-a^3*c),a^5*(b-a^3*c),a^6*(b-a^3*c),a^7*
(b-a^3*c),a^8*(b-a^3*c)]
(%i150) stemlist(y2,[],[],"1st-Order LCCDE with Initial Conditions",[],0);
Frequency Response Magnitude in dB
(%i151) freqz([1],[1,-6/10],[],[],"ampl_dB","1st-Order LCCDE Frequency Response",
[],[]);
Frequency Response Phase in Radians
(%i152) freqz([1],[1,-6/10],[],[-1,1],"phase","1st-Order LCCDE Frequency Response",
[],[]);
```

$$y[n] = \frac{3}{5}y[n-1] + x[n] \xrightarrow[\text{LCCDE}]{\text{solve}} h[n] = \left(\frac{3}{5}\right)^n u[n] \xrightarrow{\mathcal{F}} \frac{1}{1-\frac{3}{5}e^{-j\omega}} \xrightarrow{z} \frac{1}{1-\frac{3}{5}z^{-1}}, |z| > \frac{3}{5} \quad (16)$$

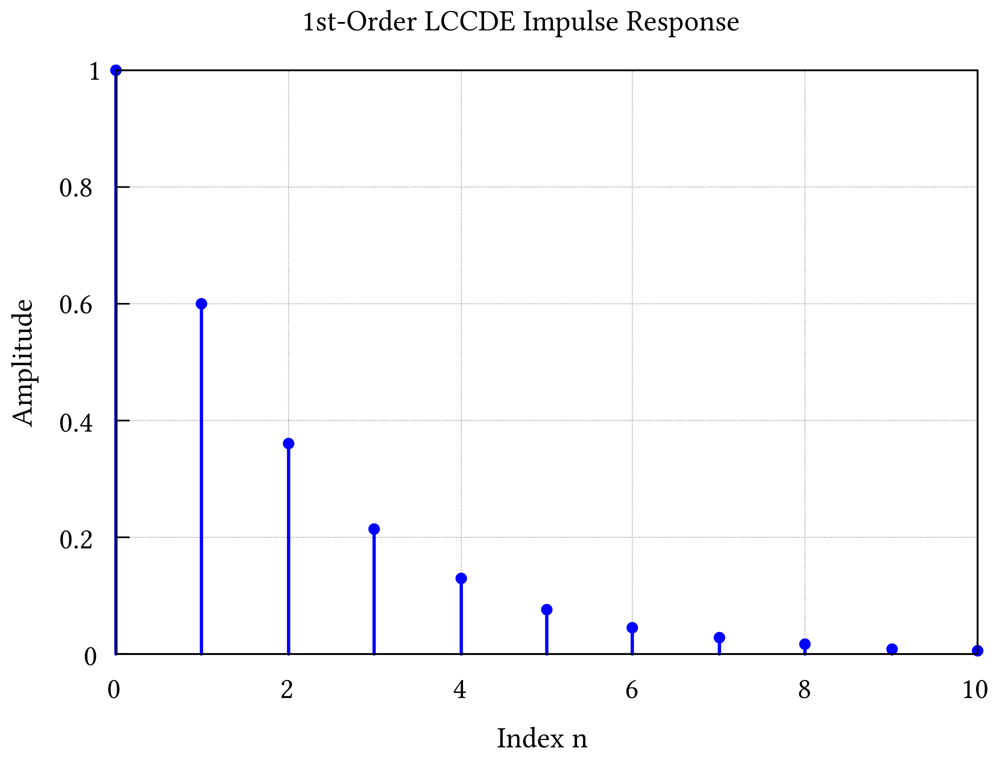

Figure 3: The first-order LCCDE impulse response as a `stem plot`.

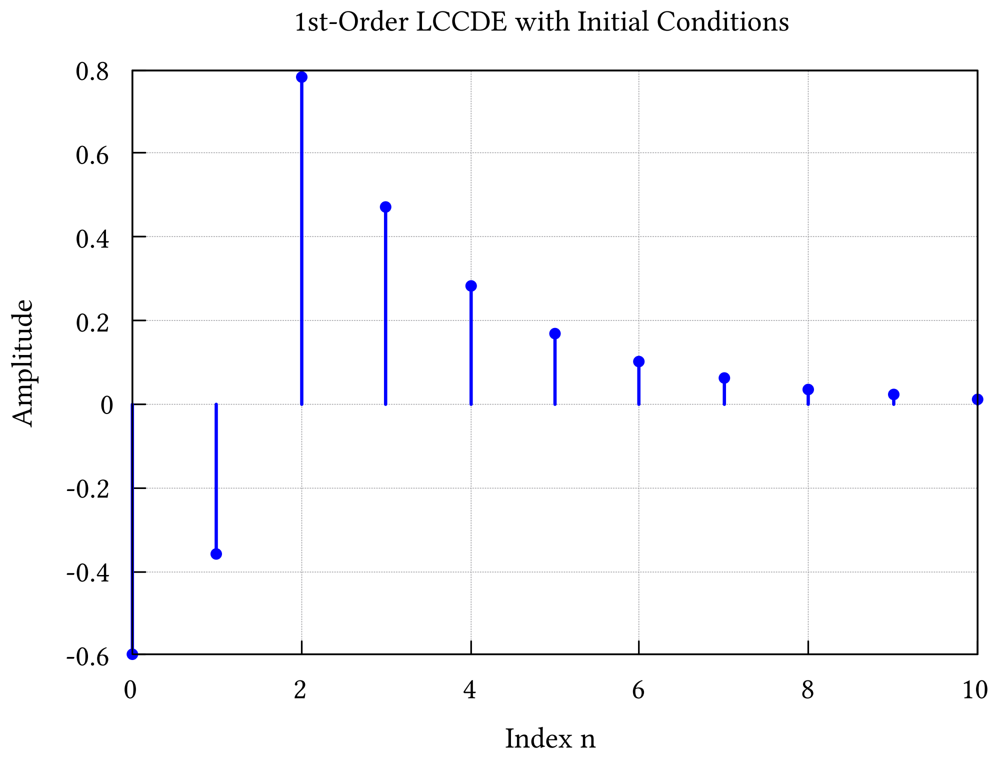

Figure 4: Response with non-zero initial conditions as a `stem plot`.

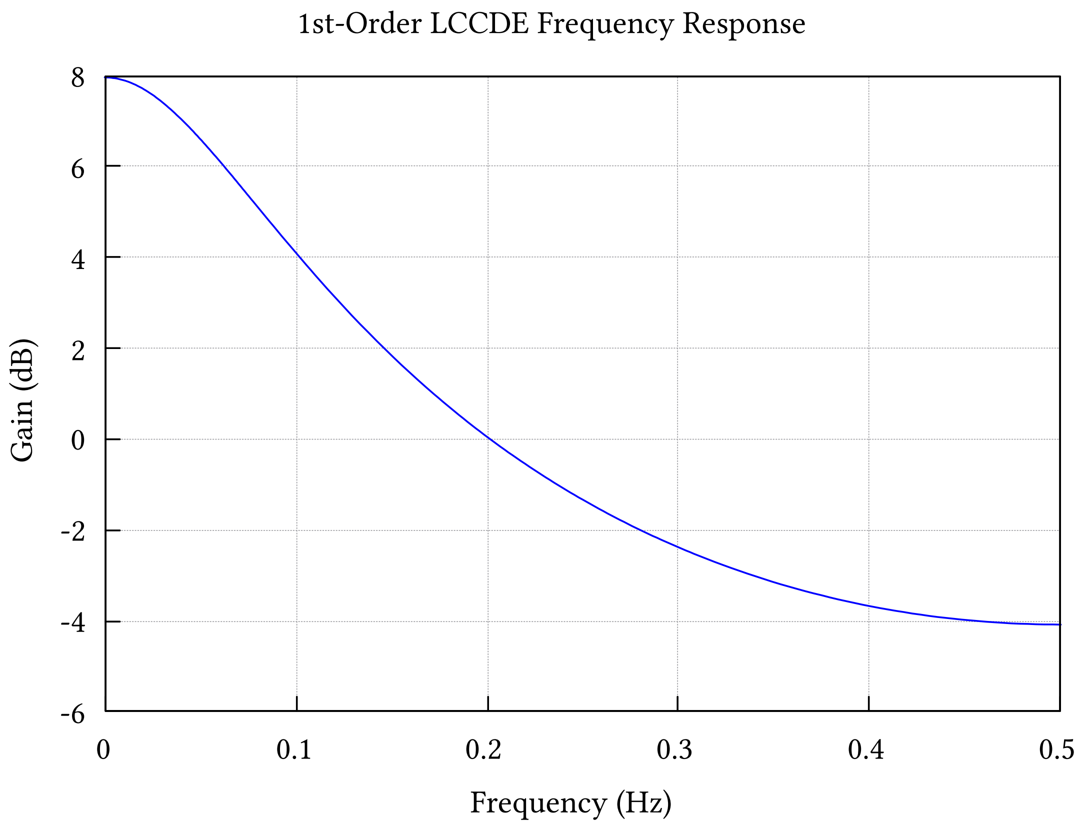

Figure 5: Frequency response magnitude as `line plot`.

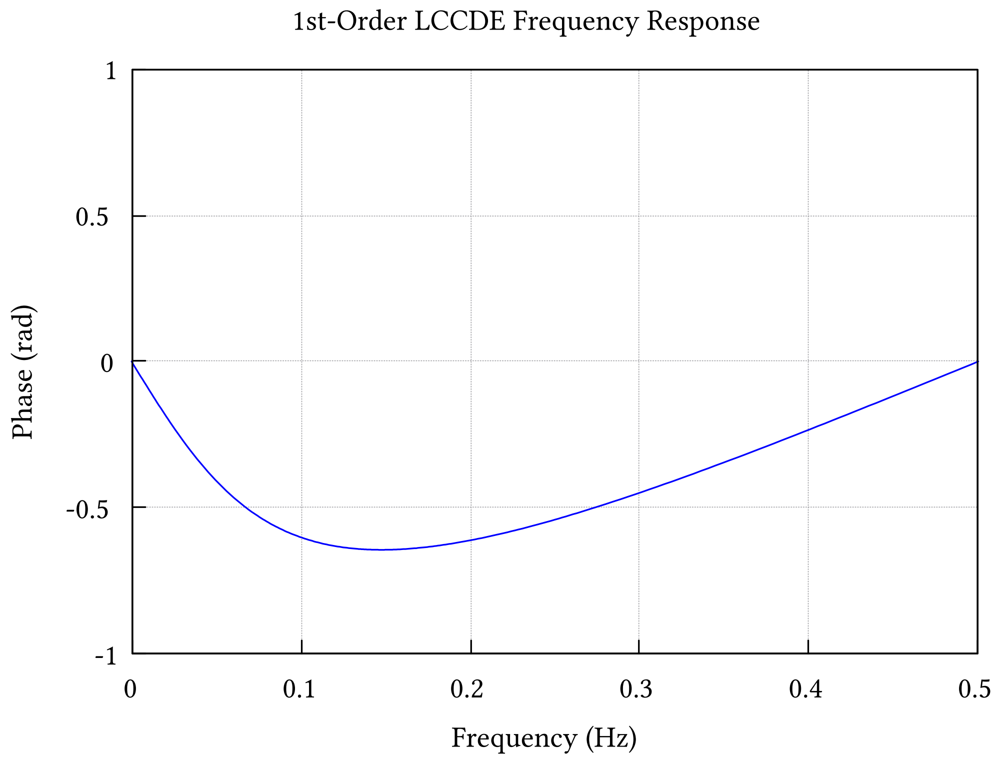

Figure 6: Frequency response phase as `line plot`.

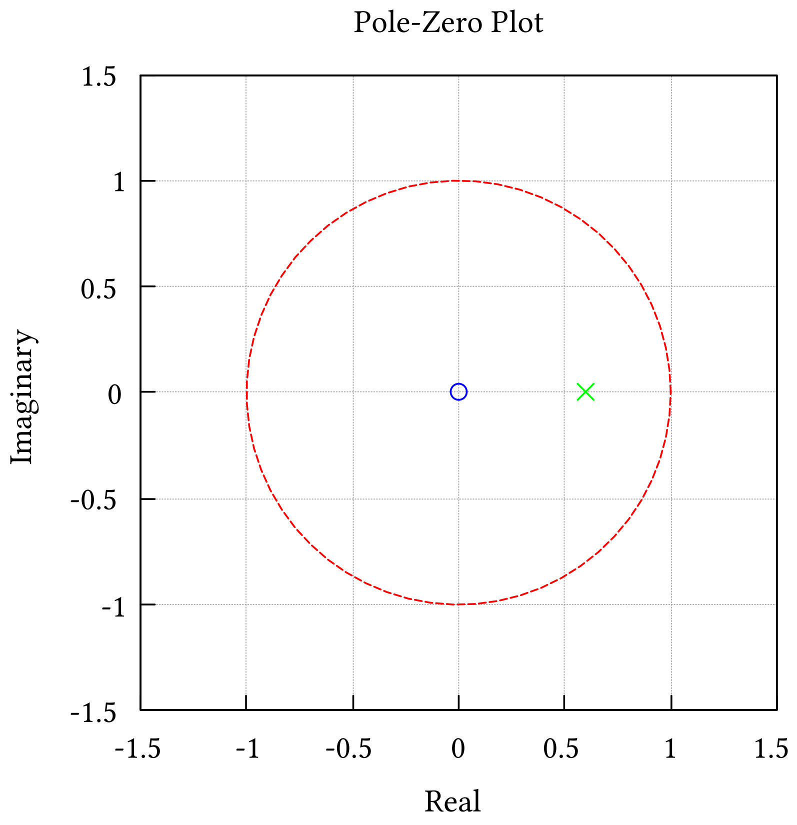

Figure 7: The pole-zero plot of $\frac{1}{1-\frac{3}{5}z^{-1}} = \frac{z-0}{z-\frac{3}{5}}$ using `zplane`.

3.4. Convolving Two Finite Duration Sequences

Consider two sequences of `ones()`. One has duration 10 and the other duration 15.The length of the convolution is known to be $L_{y_4} = 10 + 15 - 1 = 24$. See https://faculty.uccs.edu/mwickert/wp-content/uploads/sites/58/2024/08/convolution_extra.pdf for more details.

Convolving two rectangular pulses of different lengths

```
(%i163) y4:listconv(ones(10),ones(15));
(y4)  [1,2,3,4,5,6,7,8,9,10,10,10,10,10,10,9,8,7,6,5,4,3,2,1]
(%i165) stemlist(y4,[],[],"10 Ones Convolved with 15 Ones",[],0);
```

$$y_4[n] = x_{10}[n] * x_{15}[n] = \begin{cases} \sum_{k=0}^{9} x_{10}[k]x_{15}[n-k], 0 \leq n \leq 23 \\ 0, \quad \text{otherwise} \end{cases} \tag{17}$$

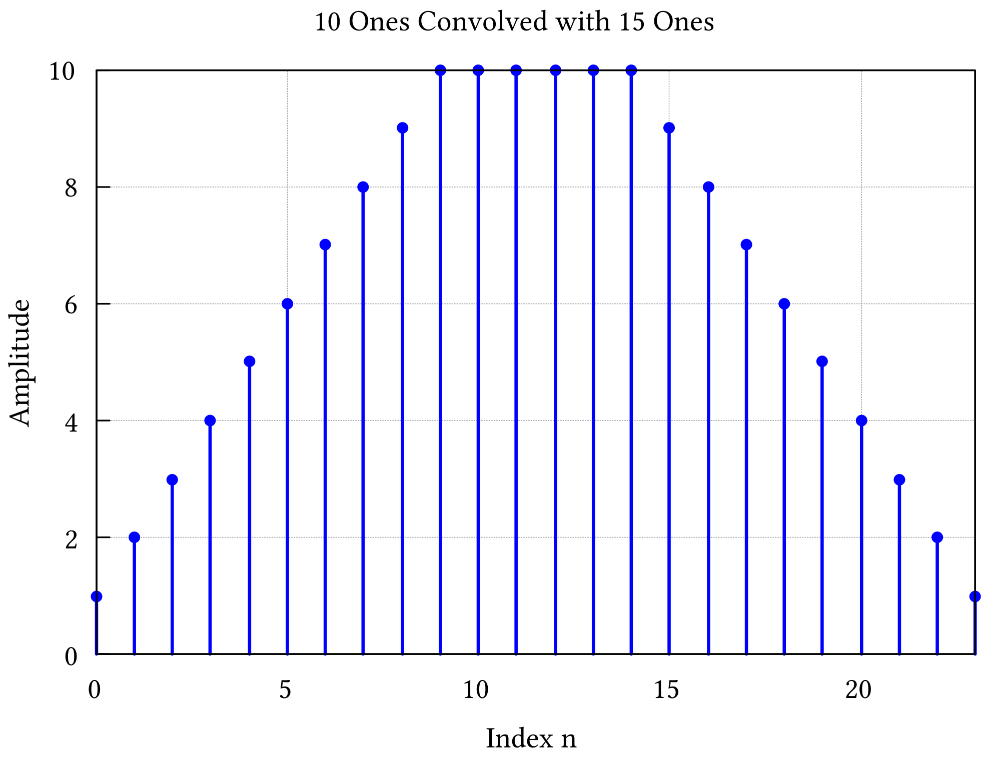

Figure 8: Trapezoid waveform from convolving different length rectangles.

3.5. Upsampling A Sequence and Simple Linear Interpolation

Here we consider a multi-rate signal processing system that consists of an upsample by four operation followed by a filter performing linear interpolation. Upsampling a signal stream by the factor L will insert $L - 1$ zero samples between the input samples. To make the filter following the upsampler perform linear interpolation, we design a triangular shaped impulse response of the form:

$$h[n] = \left\{\frac{1}{L}, \frac{2}{L}, ..., \frac{L-1}{L}, 1, \frac{L-1}{L}, ... \frac{2}{L}, \frac{1}{L}\right\}, 0 \leq n \leq 2L - 2 \tag{18}$$

In the example the input is a unit amplitude low frequency sinusoid sequence so it is easy to follow the action of the upsample by four and the linear interpolation of a 7-tap FIR filter $(2 \cdot 4 - 1)$.

Upsampling by 4 followed by a simple Linear Interpolation Filter

```
(%i178) x1:makelist(sin(2*%pi*n/50),n,0,10)$
(%i179) x2:upsample(float(x1),4)$
(%i180) stemlist(x2,[],[],"Upsample by 4",[],0);
<> Design a simple linear interpolation FIR filter for L = 4.
(%i181) b_up4:[1,2,3,4,3,2,1]/4;
(b_up4) [1/4,1/2,3/4,1,3/4,1/2,1/4]
(%i182) x2_interp:filt(b_up4,[1],x2,[],[])$
(%i183) stemlist(x2_interp,[],[],"Upsample by 4 and Linear
                                   Interpolate Samples",[],0);
<> Create longer test signals for spectral analysis using psd()
(%i184) x1L:makelist(sin(2*%pi*n/20),n,0,1000)$
(%i185) x2L:upsample(float(x1L),4)$
(%i186) length(x2L);
(%o186) 4004
(%i187) psd(x2L,1024,1.0,[-60,-10],[],[],"Upsample by 4 Sinusoid",true,[],false);
(%i188) x2L_interp:filt(b_up4,[1],x2L,[],[])$
(%i189) psd(x2L_interp,1024,1.0,[-100,10],[],[],"Upsample by 4 Sinusoid with Linear
Interp",true,[],false);
<> Spectral line height following upsample includes 1/2 from single line of
   two-sided sinusoid spectrum.
<> The factor of 1/4 is the amplitude loss due to the upsample itself, which is
   also spread the spectrum over four locations on the [0, 1/2] frequency axis.
(%i190) 20*log10(1/2*1/4,[]);
(%o190) -18.061799739838868
<> Spectral line height following interpolation filter. The passband gain (~DC
gain) of the interpolation filter (lowpass shape) is the sum of the filter
coefficients.
(%i191) 20*log10(sum(b_up4[k],k,1,7)*1/2*1/4,[]);
(%o191) -6.020599913279623
```

$$x_1[n] = \sin(2\pi n/50), 4), 0 \le n \le 10 \tag{19}$$

$$x_2[n] = \text{upsample}(x_1[n], 4), 0 \le n \le 41 \tag{20}$$

$$x_{2,\text{interp}}[n] = \text{filt}\big(b_{\text{up}}[n], [1], x_2[n]\big), 0 \le n \le 41, \text{where} \tag{21}$$

$$b_{\text{up}}[n] = \left\{\frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1, \frac{3}{4}, \frac{1}{2}, \frac{1}{4}\right\}, 0 \le n \le 6 \tag{22}$$

The output of the upsampler is shown Figure 9 where the sinusoid is increasing in amplitude from the positive zero crossing at index $n = 0$. The insertion of the 3 zero samples is evident. The output from the interpolation filter is shown in Figure 10. Here we see the zero samples nicely filled in. Is linear interpolation really that good? More on this later.

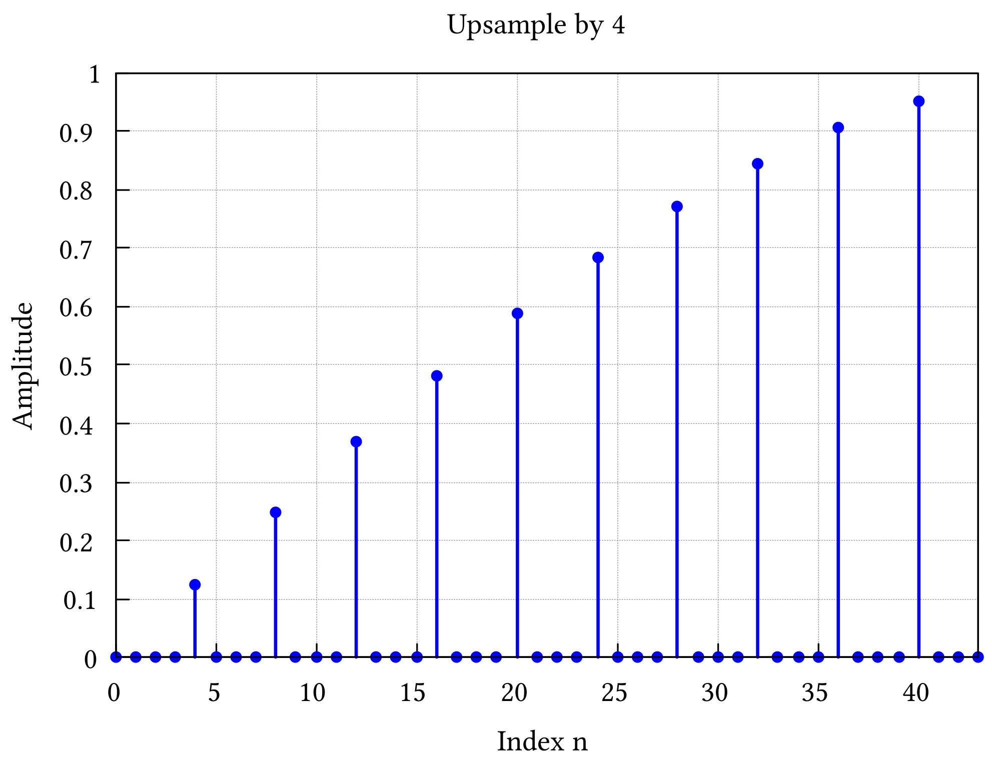

Figure 9: Ten samples of the sinusoid sequence $\sin(2\pi n/50)$ upsampled by the factor $L = 4$.

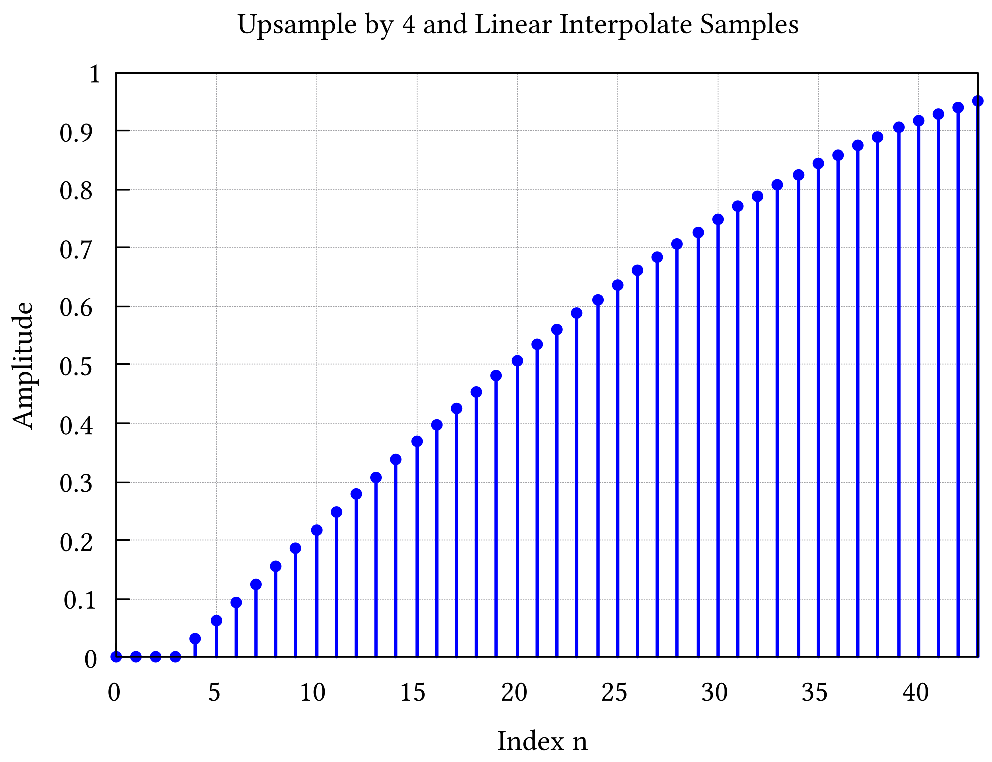

Figure 10: Apply a 7-tap FIR linear interpolation filter to the upsampled sequence.

Next we view the results in the frequency domain using a power spectral density estimate from longer length test signals. The function `psd()` has a built-in plotting capability. In the code see lines `(%i187)` and `(%i189)`. The `psd()` scaling mode is set to `true`, meaning the amplitude values in dB are approximately correct for sinusoids. In Figure 11 we see the upsampling zero insertion results in spectral replication at multiples of $\omega = \pi/L$ or with $f_s = 1$ multiples of f_s/L, e.g., spectral line clusters centered on $\left\{0, \frac{1}{4}, \frac{1}{4}, \ldots\right\}$ Hz. The spectrum of a single sinusoid will give a pair of lines at $\pm f_o$ of amplitude $A^2/2 = 1/2$ or in dB `20*log10(1/2,[])=-6.02` dB. With upsampling the amplitude is reduced by four so the amplitude of all four lines is −18.06 dB. The spectral lines at $\left\{\frac{1}{4}, \frac{1}{2}, \frac{3}{4}\right\}$ Hz are the *spectral images* that pure upsampling produces. The ideal lowpass interpolation has bandwidth $f_c = 1/8$ Hz. The frequency response of the linear interpolation filter, shown in Figure 12, on approximates the ideal filter. The passband gain of the filter is not flat and it has no-zero above f_c. This is disappointing considering how well the linear interpolation resul appears in Figure 10. In Figure 13 the spectrum following interpolation is shown. The desired spectral line at $f = \frac{1}{50} = 0.02$ is large, but spectral lines at eth image frequencies also pass through the filter suppressed in this case by 50 dB. The spectral line at 1/50 has increased amplitude as a result of the filter gain at 1/50. The filer gain at DC, close to 0.02 Hz, is just the sum of the filter coefficients = 20*log10(sum(b_up4[k],k,1,7) $\cdot$ 1/2 $\cdot$ 1/4, []) $= -6.02$ dB and concurs with psd estimate.

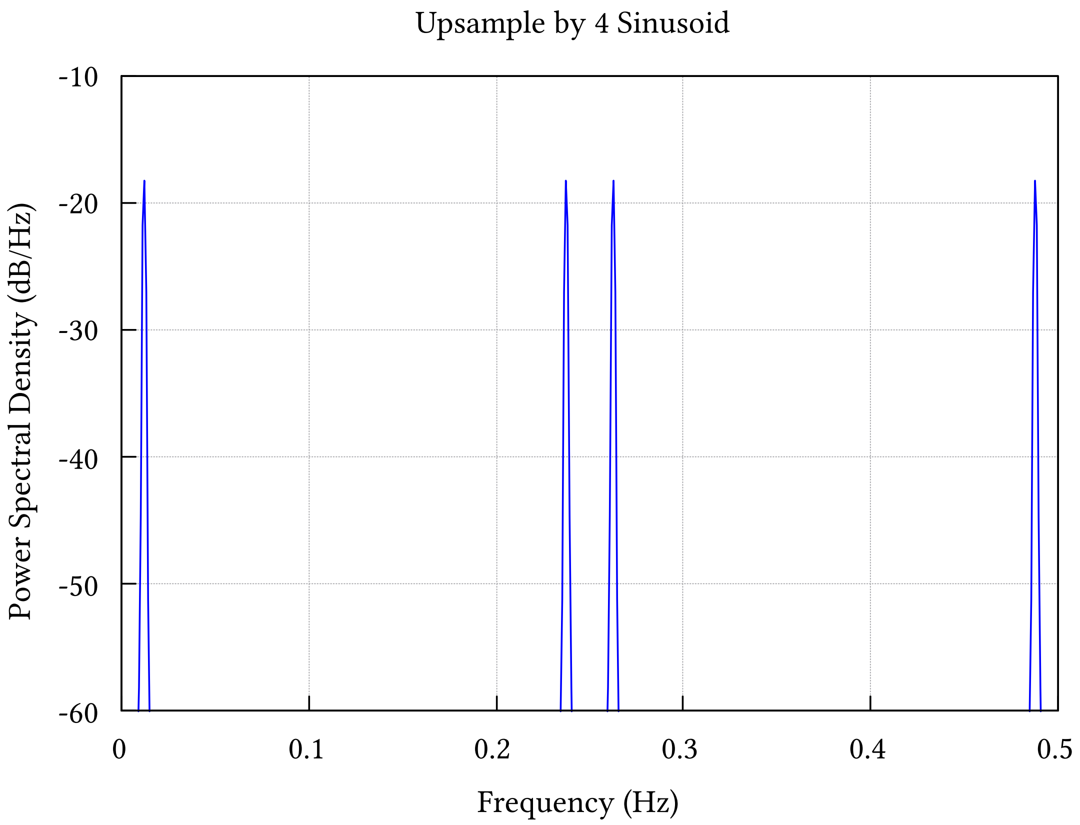

Figure 11: Estimate the power spectrum of the upsampled sinusoid using 4001 samples.

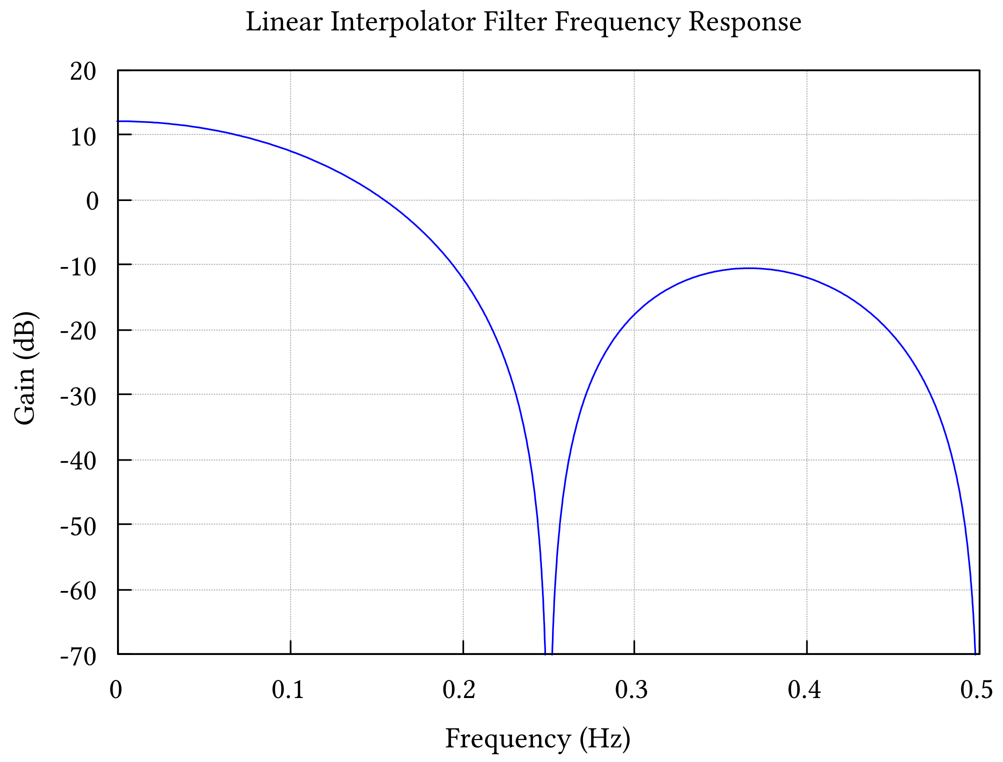

Figure 12: The 7-tap linear interpolation frequency magnitude in dB.

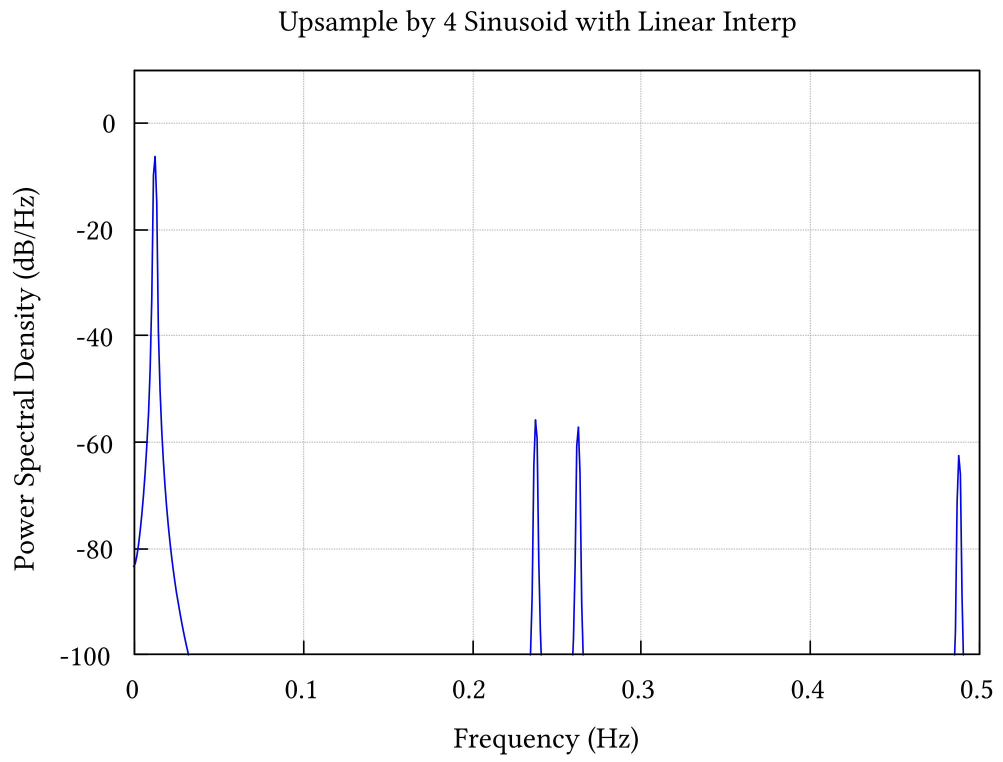

Figure 13: Estimate the power spectrum of the upsampled and filtered output.

4. Comm Tools Examples

4.1. Generate an NRZ Data Waveform with a Rectangular Pulse Shape plus Noise

Here we generate a 500 data bit NRZ waveform, `x_bb`, so we can later see the *eye plot* following *matched filtering*. For simple waveform plots we create a subset list using the Maxima `makelist()` function. Inside this function the data bits are obtains using the stats function `random_bernoulli(1/2,N_bits)` transformed from zero one values to ± 1 values, which is typical in digital communications. In *complex baseband* modeling of digital communications it is typical represent signals that travel on a radio frequency carrier to be represented as a complex sequence or list. For *binary phase-shift keying* (BPSK) the binary information is carried in the real-part of the complex baseband waveform while the imaginary part is zero. To represent a received waveform containing an additive Gaussian/Normal noise (AWGN) random process we use the function`cpx_awgn()` which adds Normal random variates to the real and imaginary parts of the input waveform using `random_normal(0, std_val, N_samps)`, with the standard deviation chosen to represent the desired energy-per-bit to noise-power spectral density ratio in dB, i.e., $(E_s/N_0)_{\mathrm{dB}}$. For binary signaling bits and symbols are equivalent. To optimally recover the binary data from the waveform in AWGN we cam *matched filter* the noisy received using a filter having a rectangular pulse shape impulse response. In this case the transmit pulse shape is 10 samples long so the matched filter impulse response is also 10 samples long. The matched filter output needs to be sampled at the maximum *eye opening* once per bit, or every 10 samples in this example. The eye plot is obtained using `eye_plot()` is an overlay plot of all the bit transitions in the `y_mf` waveform, plotted over two bits or 20 samples, give plot the *eye* appearance. When the full complex waveform is sampled at the maximum eye opening we obtain using `constellation` a plot of the signal *decision statics* for sign detection/classification of the receive bit being +1 or −1. The scatter points of the constellation plot are sample points of a conditionally 2D Gaussian probability density function (PDF).

Using NRZ_bits() and cpx_awgn() from \`CommTools.mac\` and Plotting functions from DSPTools.max and CommTools.mac

```
(%i58)  x_bb:NRZ_bits(500,10,"rect",0.35,[])$
(%i59)  plotlist(makelist(x_bb[k],k,1,500),"10 Samples Per Bit","NRZ Bits",
      "Rectangle Pulse NRZ Waveform: 50 Bits",[-1.1,1.1], 0);
(%i60)  r_bb:cpx_awgn(x_bb,20,10)$
"Standard Deviation: "0.223605008888844143
(%i61)  plotlist2(realpart(makelist(r_bb[k],k,1,500)),makelist(x_bb[k],k,1,500),
      "10 Samples per Bit","NRZ Bits","NRZ Bits with Eb/N0 = 20 dB",[-2,2], [0,0]);
(%i62)  y_mf:filt(ones(10)/10,[1],r_bb,[],[])$
(%i63)  plotlist(makelist(realpart(y_mf[k]),k,1,500),"10 Samples Per Bit",
      "NRZ Bits","Matched Filtered NRZ Waveform",[-1.5,1.5], 0);
(%i64)  eye_plot(realpart(y_mf),30,20);
"Slices Plotted: "166
(%i65)  constellation(y_mf,10,9,20,[]);
497
```

$$x_{\mathrm{bb}}[n] = \sum_{i-1}^{500} d_i p(n - i \cdot 10), \text{where } d_i \text{ is} \pm 1 \text{ rand int and } p[n] \text{ is 10 samp rect pulse} \quad (23)$$

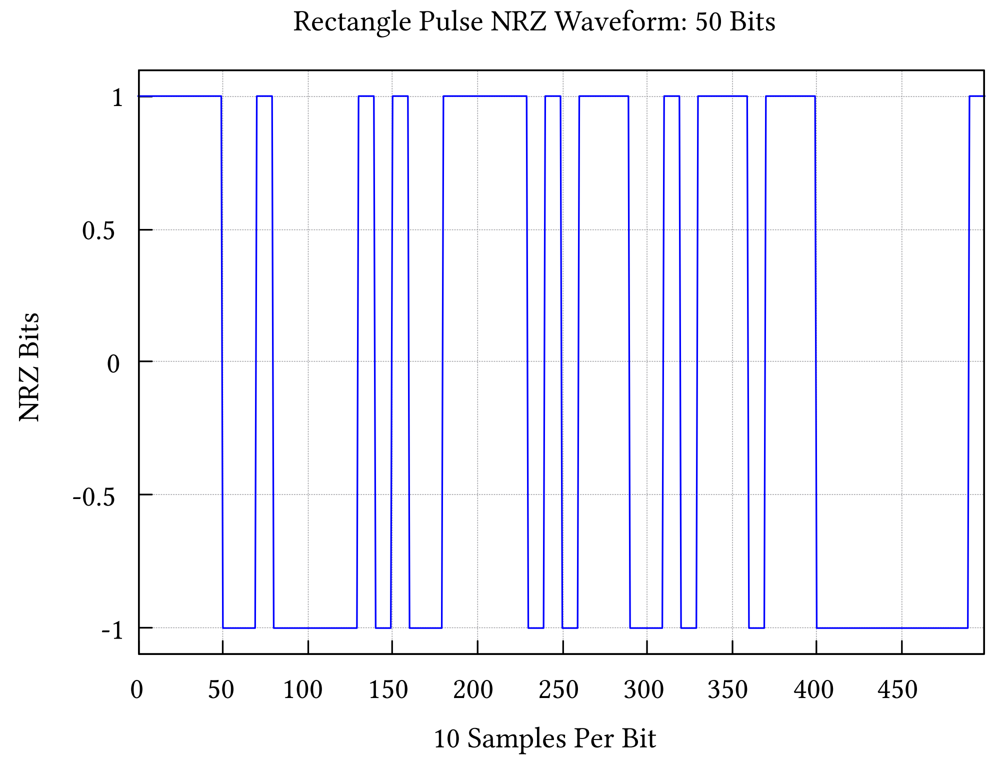

Figure 14: NRZ data bits waveform using a rectangular pulse shape and 10 samples per bit.

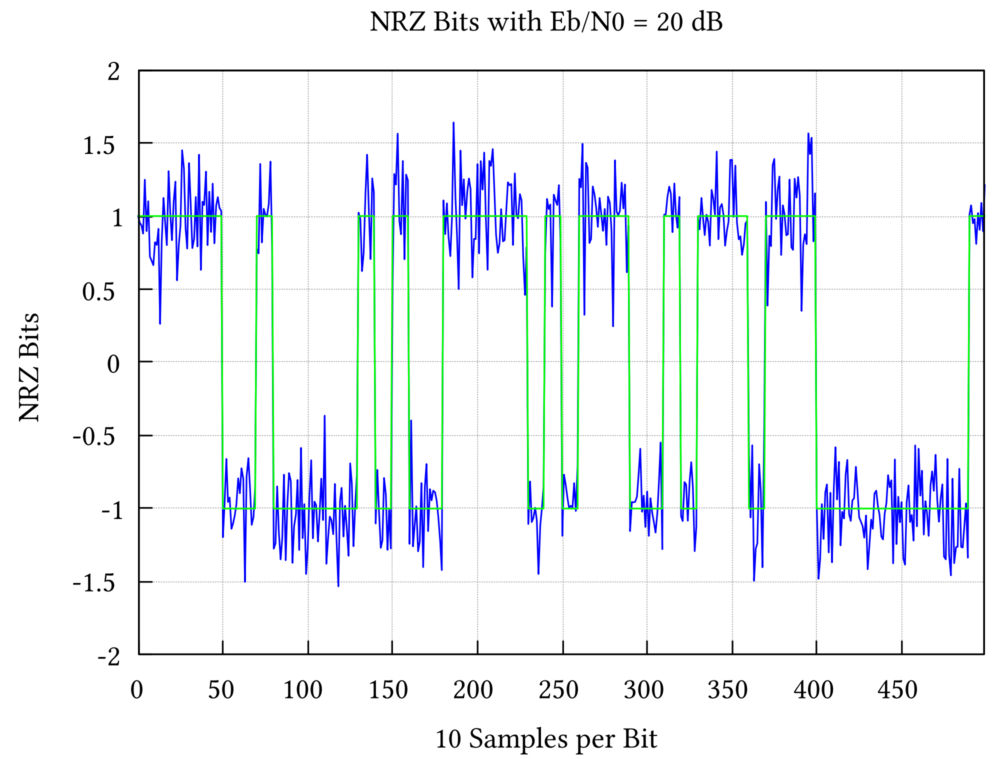

Figure 15: Real part of NRZ data bits in complex additive white Gaussian noise.

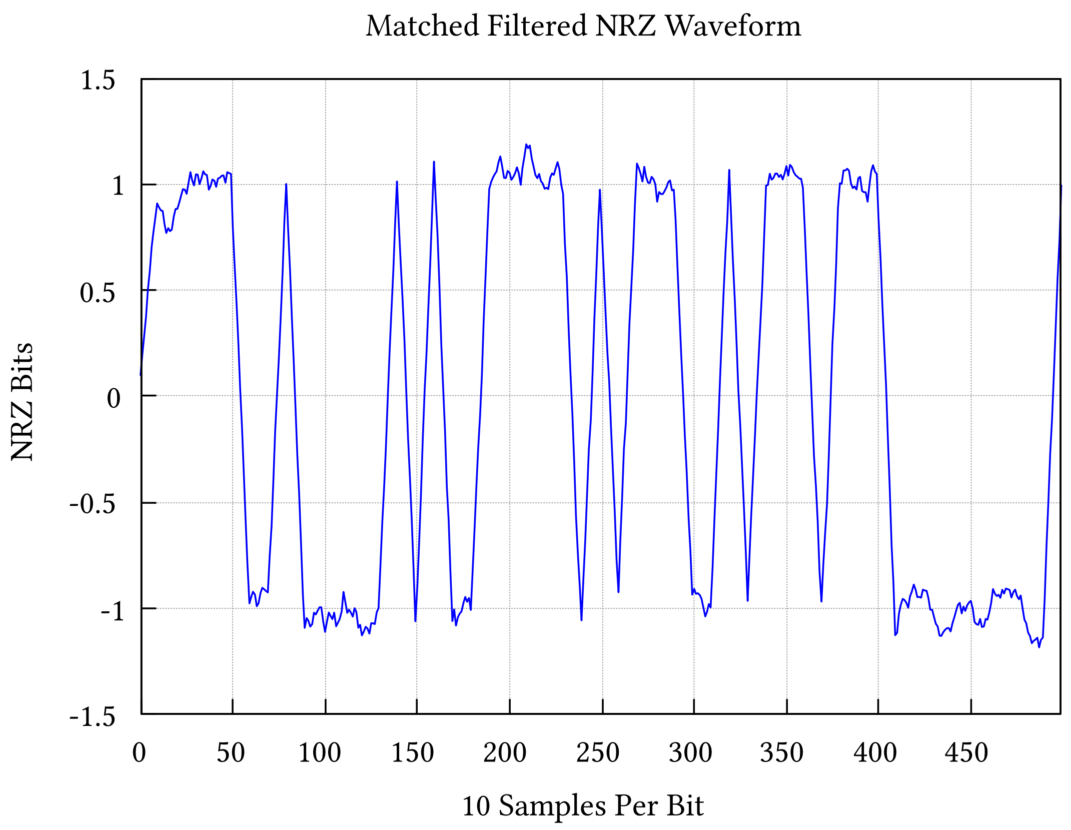

Figure 16: Match-filtered noisy NRZ waveform (real-part).

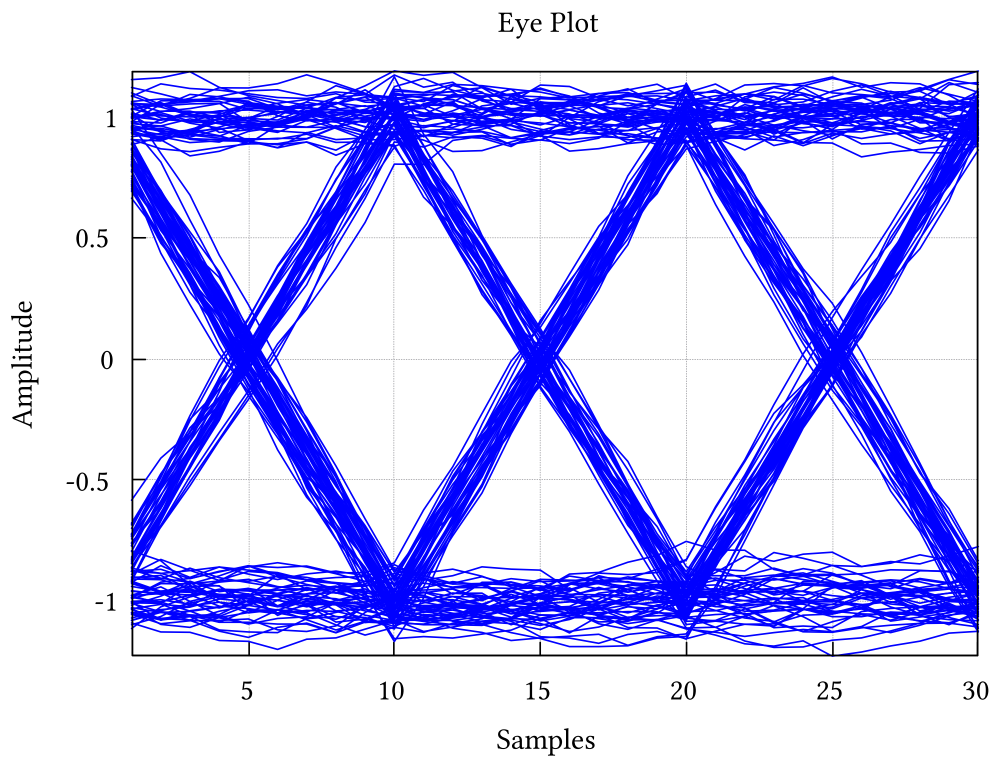

Figure 17: Eye plot of match-filtered noisy NRZ waveform (real-part).

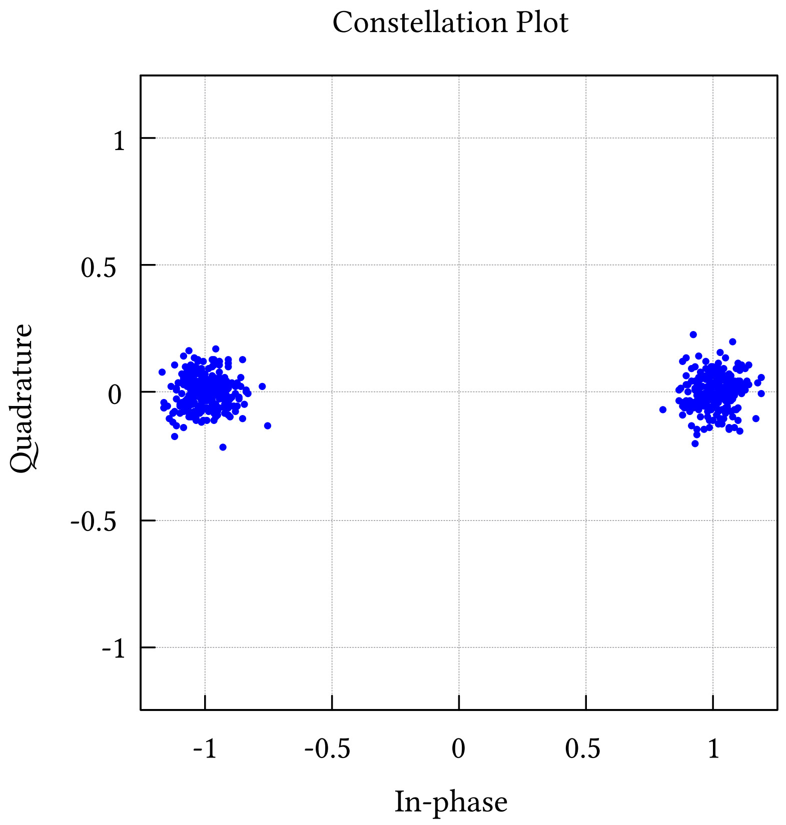

Figure 18: IQ-plane constellation plot of the complex match-filtered noisy NRZ waveform.

5. Filter Tools Examples

In this section we provide two examples that function in the module `filter_tools.mac`. This first is continuation of the upsample by 4 example. The second explore the IIR notch filter on an audio test signal with jamming.

5.1. An Improved Interpolation Filter for the Upsample by 4 Example

The simple linear interpolation used in the upsample by 4 example earlier produced seemingly good looking time domain results, but the psd plot revealed spectral leakage at image bands ≈ 50 dB down. We design a lowpass filter design of specified passband and stopband, with the stopband attenuation set to 80 dB. The function `firwin_kaiser_lpf(f_pass, f_stop, d_stop, fs, [n_bump)]` can do this for us. To keep the filter order (number of taps) from getting out-of-hand we create a transition band by setting $f_{\text{pass}} = 1/8 - 0.05$ and $f_{\text{stop}} = 1/8 + 0.05$. The total transition bandwidth is 0.1 Hz and can be adjusted as needed. In the design the design has $N_{\text{taps}} = 52$, which is reasonable.

An firwin Interpolation Design for the Earlier Upsample by 4 Example

```
(%i258) b_L4:firwin_kaiser_lpf(1/8 - 0.05, 1/8 + 0.05, 80, 1.0, 0);
(b_L4)  [3.0823657133829874*10^-5,3.440370104460139*10^-5,
        -7.043831398681369*10^-5,-3.046304890264764*10^-4,
        -5.021826000916669*10^-4,-3.2343651037778823*10^-4,
         4.8117666672903016*10^-4,0.0016687860721234914,
         0.002329721319765943,0.0013154297086413486,
        -0.0017572785273746575,-0.005570915085770377,-
         0.007209245521049741,-0.0038171086408114348,
         0.004830215115116467,0.01464026302319773,
         0.018279613963765578,0.00942788925327944,
        -0.011746691737271973,-0.035515858392871955,
        -0.04499589025773563,-0.024127267973041936,
         0.03247054768373759,0.1135522251377183,
         0.1935874330050354,0.24329241574212157,
         0.24329241574212157,0.1935874330050354,
         0.1135522251377183,0.03247054768373759,
        -0.024127267973041936,-0.04499589025773563,
        -0.035515858392871955,-0.011746691737271973,
         0.00942788925327944,0.018279613963765578,
         0.01464026302319773,0.004830215115116467,
        -0.0038171086408114348,-0.007209245521049741,
        -0.005570915085770377,-0.0017572785273746575,
         0.0013154297086413486,0.002329721319765943,
         0.0016687860721234914,4.8117666672903016*10^-4,
        -3.2343651037778823*10^-4,-5.021826000916669*10^-4,
        -3.046304890264764*10^-4,-7.043831398681369*10^-5,
         3.440370104460139*10^-5,3.0823657133829874*10^-5]
(%i259) length(b_L4);
<> Filter taps 52 (51st-order polynomial)
(%o259) 52
(%i260) zplane(b_L4,[],[]);
(%i265) freqz(b_L4,[1],[],[-90,5],[],"L = 4 firwin Filter Frequency Response",[],
[]);
(%i261) x2L_L4:filt(b_L4,[1],x2L,[],[])$
(%i263) psd(x2L_L4,1024,1.0,[-100,10],[],[],"Upsample by 4 Sinusoid  with firwin
Interp",true,[],false);
```

$$b_{L4}[n] = \text{firwin\_kaiser\_lpf}\left(\frac{1}{8} - 0.05, \frac{1}{8} + 0.05, 80, 1.0, 0\right), \text{transition band} = 0.10 \text{ Hz} \quad (24)$$

We begin by looking at the filter pole-zero plot in Figure 19. The zero on the unit circle form the filter stop band while the zero *quadruplets* conjugate reciprocal with the unit circle design the passband. The overall configuration of the zeros implies that the filter has *linear* phase.

The filter frequency response magnitude, shown in Figure 20, shows a nice flat passband (if zoomed small ripple would appear), relatively narrow transition band, and stop band ripple that drops off. These are the characteristics of the classical windowed FIR design.

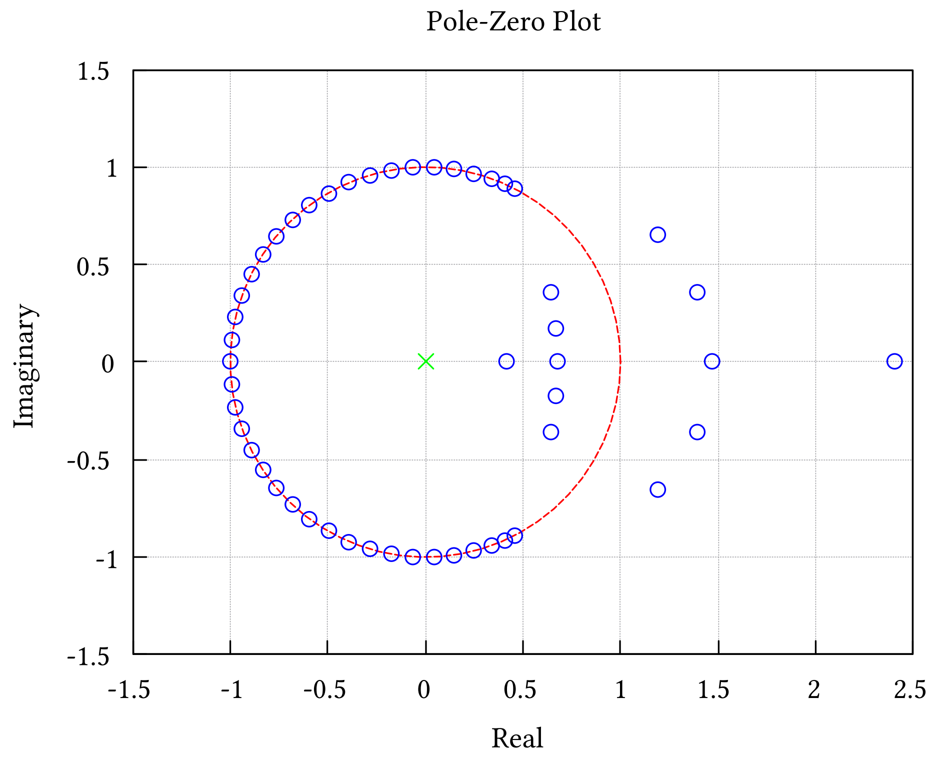

Figure 19: Pole-zero plot of the 52-tap interpolation filter.

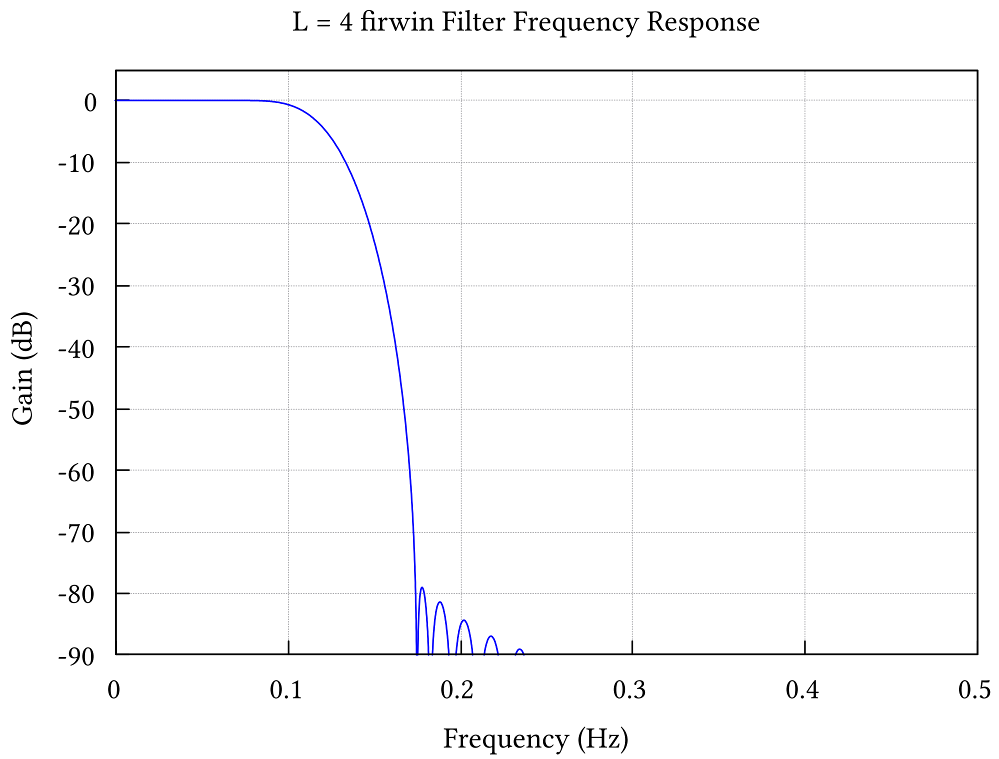

Figure 20: The frequency response magnitude in dB.

Finally looking the interpolator output in Figure 21, we see the expected spectral of the input signal at 1/50 = 0.02 Hz. The height of this line is lower than with the linear interpolator because the filter passband gain is unity in this case. The critical stopband frequency of 0.125 + 0.05 = 0.13 Hz the filter frequency response is just passing through −80 dB, We can adjust the final design by adjusting `n_bump` above of below zero. As it stands now the design now leakage harmonics at the image frequencies are not visible an $\approx$ 80 dB down relative to the desired spectral line output.

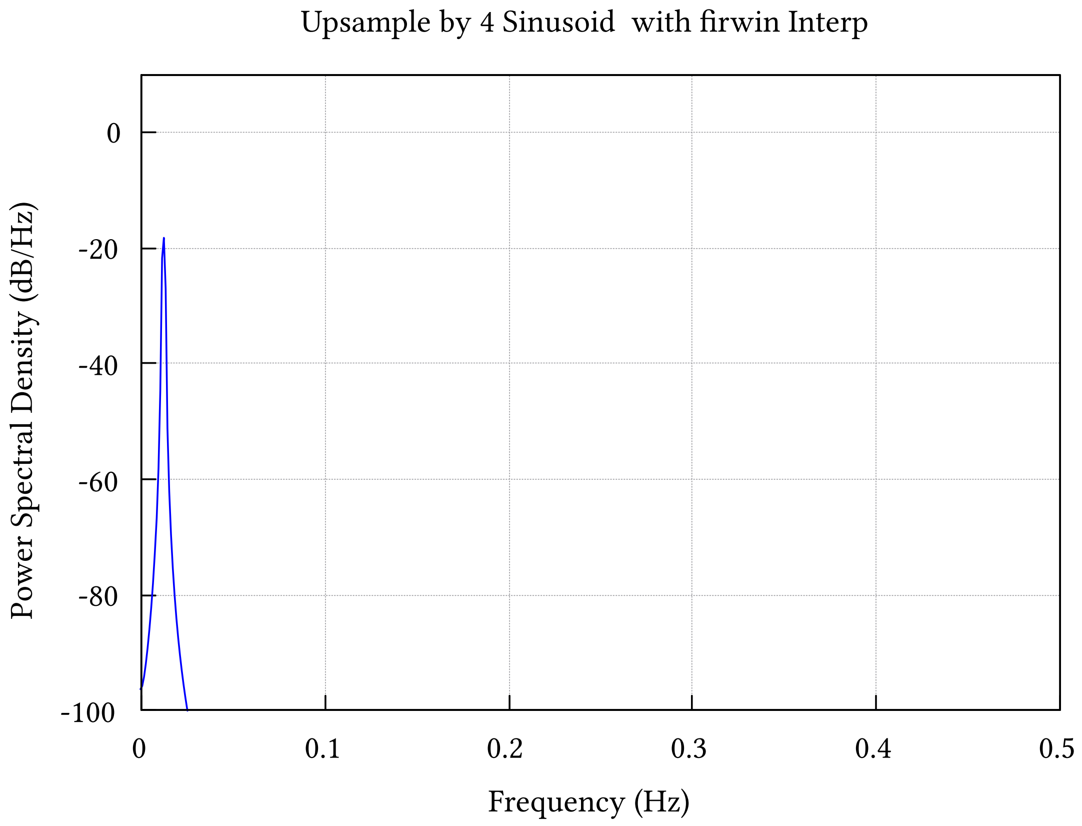

Figure 21: The psd of the single sinusoid spectrum following interpolation.

5.2. An IIR Notch Filter Design for and Audio Signal with Tone Jamming

Here we consider a speech signal that has a single sinusoid jamming tone added to it. In the z-plane the IIR notch filter places both complex conjugate zeros on the unit circle and complex conjugate poles on the same radial as the zeros but at radius $r < 1$, to sharpen the notch frequency response so that looks like the cross-section of a thumbtack. Notch filter pole-zero plot is shown in Figure 22. The angle the zeros and poles make to the positive real axis is $\pm 2\pi f_0/f_s$. The frequency response magnitude in dB is shown in Figure 23.

The speech signal was created using a laptop microphone function in the Python package scikit-dsp-comm. The file started as a `*.wav` file but was later saved to a `*.txt` file for easy input into Maxima using `x:read_list("my_record2.txt"," ")$ The tone jammiing signal was added in Maxima` to form received signal `r = x + tone_jam`. The `filt()` was then used to filter the signal using a second-order IIR difference equation.

Designing an IIR Notch Filter to Remove a Single Tom,e Jamming Signal

```
(%i14)  x:read_list("my_record2.txt"," ")$
(%i15)  length(x);
(%o15)  40000
(%i16)  tone_jam:makelist(float(cos(2*%pi*1000/44100*n)),n,1,40000)$
(%i17)  r:x + 1/2*tone_jam$
(%i18)  plotlist(r,[],[],"Audio Samples with 1 kHz Jammer",[],0);
(%i19)  ba:fir_iir_notch(1000,44100,0.9);
(ba)  [1,-(2*cos((20*%pi)/441)),1,1,-(1.8*cos((20*%pi)/441)),0.81]
(%i20)  baf:float(ba);
(baf) [1.0,-1.9797349455598832,1.0,1.0,-1.7817614510038948,0.81]
(%i21)  zplane(rest(baf,-3),rest(baf,3),[]);
(%i22)  freqz(rest(baf,-3),rest(baf,3),[],[-100,10],"ampl_dB","1 kHz IIR Notch
Filter for fs = 44,100 Hz",[],44100);
(%i23)  rf:filt(rest(baf,-3),rest(baf,3),r,[],[])$
(%i24)  plotlist2(r,rf,[],[],"Signal with Jamming (blue) and Jamming Removed
(green)",[],[0,0]);
```

$$H(z) = \frac{1 - 2\cos(20\frac{\pi}{441})z^{-1} + z^{-2}}{1 - \frac{9}{5}\cos(20\frac{\pi}{441})z^{-1} + \frac{81}{100}z^{-2}} = \frac{1.0 - 1.97973z^{-1} + 1.0z^{-2}}{1.0 - 1.78176z^{-1} + 0.81z^{-2}} \tag{25}$$

$$r_f[n] = 1.78176r_f[n-1] - 0.81y_f[n-2] + r[n] - 1.97973r[n-1] + r[n-2], n \geq 0 \tag{26}$$

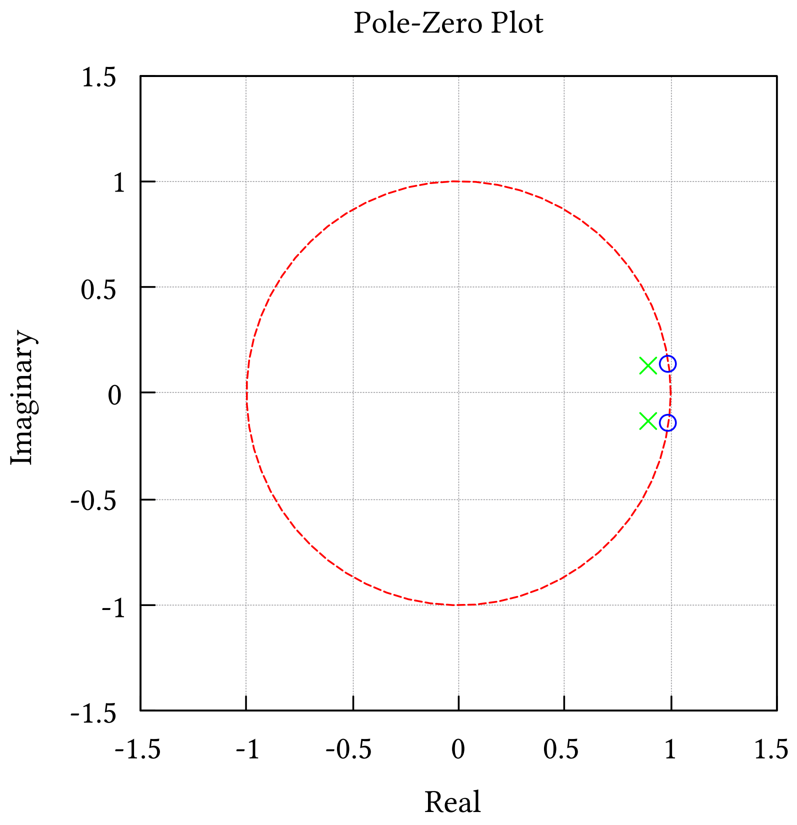

Figure 22: Pole-zero plot of the notch filter.

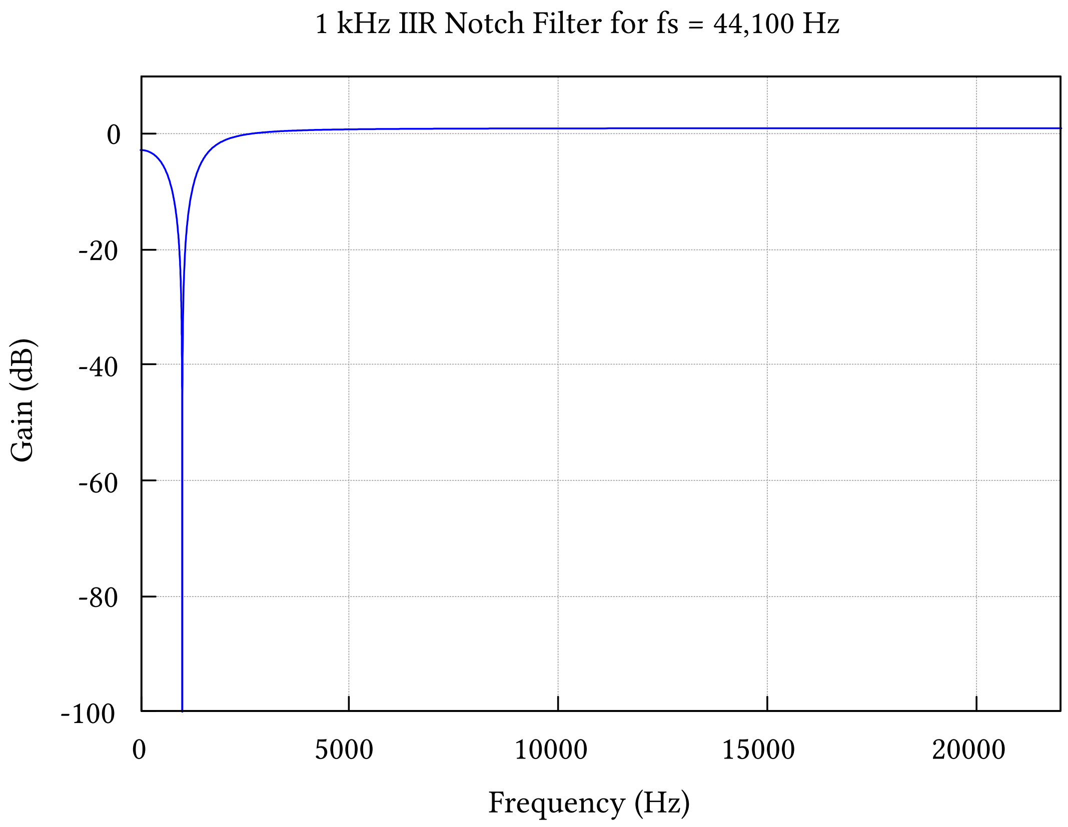

Figure 23: The frequency response magnitude of the notch filter centered on 1 kHz.

The input signal $r[n]$ is shown in Figure 24. The filtered output is shown in Figure 25. As expected for a narrowband jamming signal, the notch filter is able easily remove the jamming. With knowledge of the exact jamming frequency, say using FFT spectral analysis, it might be better to make the pole radius smaller so the notch is wider, yet some interference would leak in, but better than no excision at all taking place.

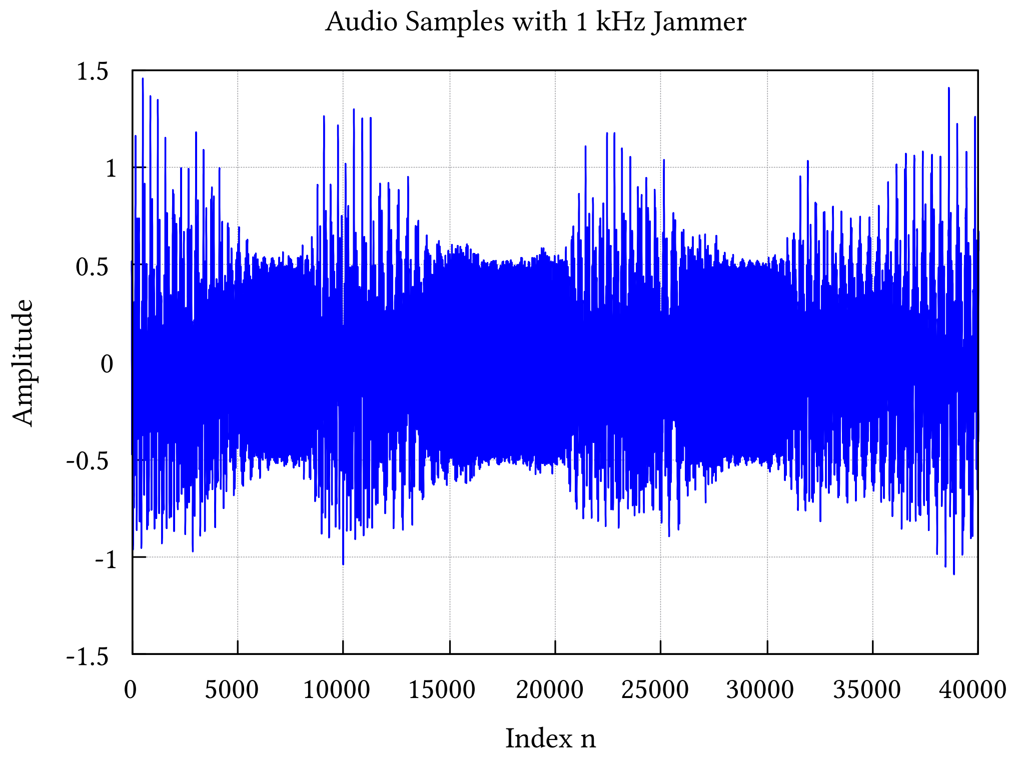

Figure 24: The signal plus tone jamming input signal.

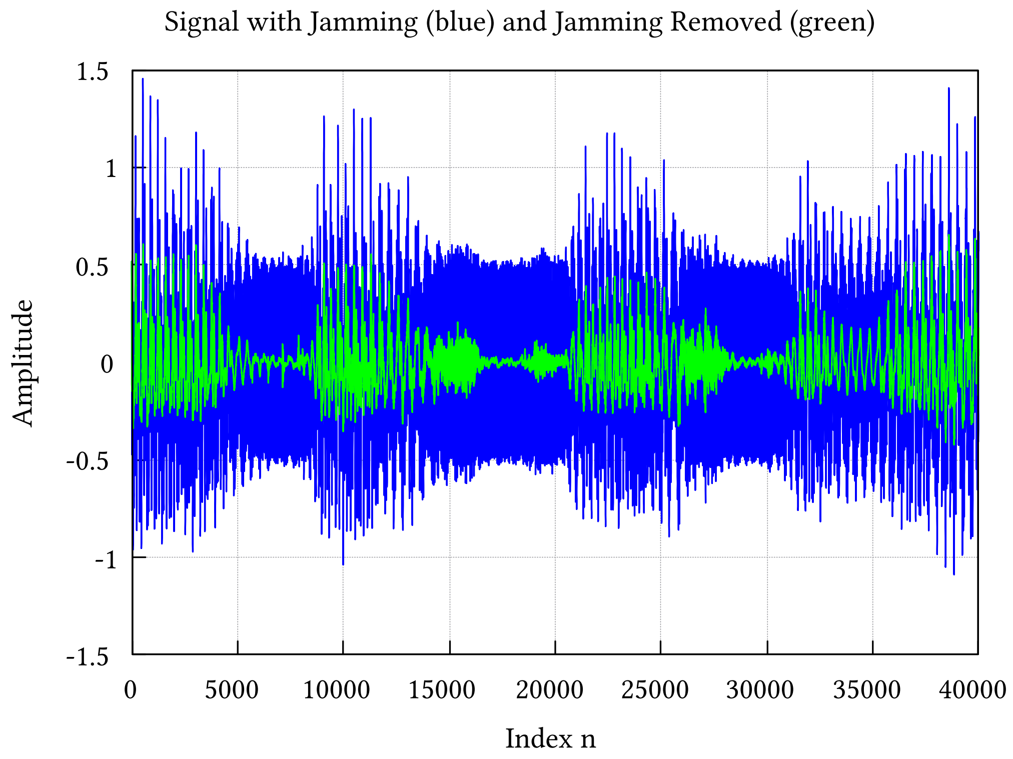

Figure 25: Filter output with jamming excised..

Bibliography

[1] M. A. Wickert, *Signals and Systems for Dummies*. For Dummies, 2013.

[2] A. V. Oppenheim and R. V. Schafer, *Discrete-Time Signal Processing, third edition*. Prentice Hall, 1999.

[3] R. Ziemer and W. Tranter, *Principles of Communications, seventh edition*. Wiley, 2014.