Laboration Exercises in Digital Signal Processing

Mikael Swartling

Department of Electrical and Information Technology
Lund Institute of Technology

revision 2014

Introduction

Introduction

The traditional way to motivate digital signal processing is the wish to process an analog signal digitally using a computer or another form of digital hardware. Today digital signal processing is cheap and fast, can be accomplished on a small area, with great flexibility. Figure 1 shows the basic principle, where a real-world analog signal is sampled to a discrete-time signal consisting of a sequence of values or samples. The number of samples produced per second in this process is determined by the sampling frequency. The discrete-time signal can then be manipulated in different ways using digital signal processing, after which it is interpolated back to an analog signal, and transmitted to a proper analog system, e.g. a loudspeaker or an antenna.

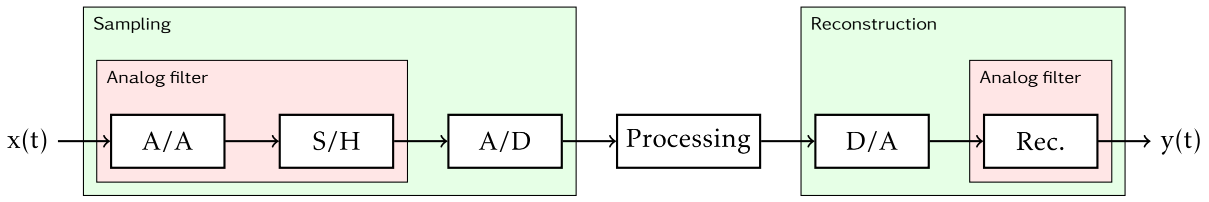


Figure 1: Sampling and reconstruction system.

Sampling

Already at an early stage of this course it is important to understand the principles and purposes of the different parts of this system. Discrete-time signals have a very fundamental property, namely they lack knowledge of at what rate they have been sampled or at what rate they will be reconstructed. Take the example of a CD disc containing a discrete-time signal. This signal does not know what time interval or sampling interval there should be between its samples, instead the CD player provides the information that the signal is to be reconstructed with 44.1 kHz. This was also known in the studio where the disc was recorded. It is often said that a discrete-time signal has distance 1 between its samples, i.e. sampling period 1 without a unit. The sampling frequency is the inverse of the sampling period, and is hence also 1 without a unit. All frequencies of the discrete-time signal are given as a fraction of the sampling frequency. These frequencies are called normalized frequencies and are between 0 and 1. It is only when the signal is reconstructed with a suitable sampling frequency that the normalized frequencies are transformed to real frequencies, since it is then one decides what units are to be used for the time axis.

Another very important property of discrete-time signals is the fact that the largest frequency that can be represented is the frequency of a signal consisting of every second sample positive, and every second sample negative, with equal amplitude. This frequency has a period of 2 samples which is why the normalized frequency is 0.5. One can try to record analog signals of faster variation, but these frequencies will not be represented correctly by the discrete signal. In fact, all frequencies of the analog signal which are above a normalized frequency of 0.5, corresponding to half the sampling frequency, will look like lower frequencies in the discrete-time signal. This phenomenon is called aliasing, and makes it impossible to map the signal back to the analog domain at its original frequency, since the aliased higher frequencies in the reconstruction process are considered to be in the interval from 0 to 0.5. As an example we can consider two analog signals of 100 Hz and 350 Hz which are sampled at 500 Hz. The signal at 350 Hz is above half the sampling frequency and cannot be represented by the discrete-time signal, instead it is considered as a signal of frequency between 0 Hz and 250 Hz, in this case $500 - 350 = 150$ Hz. If these two sampled signals are reconstructed we will get back the original 100 Hz tone, plus a tone of the aliased signal.

The solution of this problem is naturally to only allow recording of signals which can be represented correctly in the discrete-time signal. For this reason one often use an analog anti-aliasing filter, which lets through only frequencies lower than half the sampling frequency, before a signal is transformed from the analog to the digital domain with aid of a sample-and-hold circuit and an analog-to-digital converter. In this case the mapping between the analog and the digital domain is one-to-one; the conditions of the *sampling theorem* are fulfilled.

The fact that the conditions of the sampling theorem are fulfilled also means that we do not lose any information in the A/D converter, i.e. that the original analog signal can be reconstructed (interpolated) exactly from the discrete-time signal. This is accomplished by the digital-to-analog converter holding the value of each sample during a sampling interval at its output. The result is a piecewise constant signal, with stepwise changes at each new sampling interval. These stepwise changes introduce higher frequencies than half the sampling frequency. In fact all discrepancy from the original signal lies in frequencies above half the sampling frequency. For this reason one again applies an analog filter which is called a reconstruction filter in order to eliminate frequencies above half the sampling frequency, which obviously were not sent into the system. Now the analog reconstructed signal looks exactly like the original analog signal, provided that we sampled twice as fast as the largest frequency of the input signal. If this had not been done, the anti-aliasing filter would have eliminated the large frequencies of the input signal, so the condition is fulfilled in any case.

Filters

Finally some words about the concept of *filter*. A filter describes how a certain number of samples of a signal, up to the latest, shall be linearly combined in order to produce a certain output signal. The weights are put in a sequence forming a discrete-time sequence called the *impulse response* of the filter. The impulse response is a representation of an LTI system. Another description of the same system is as a difference equation. It is very important to realize that the concepts of filter and impulse response are not as strange as they first can seem. A filter is nothing but a tool in order to combine input data to obtain a desired output signal.

Let us assume you save money on a bank account each week, where $x[n]$ is the amount of money saved during week number n. We call $x[n]$ a signal, a discrete-time input signal to a system. Let us determine what output signal we desire; this will determine the system. As output signal from the system we choose, for instance, the sum of your savings the last month, i.e. the last four weeks. The difference equation for the system describes the output signal the current week as a sum of the four most recent savings where each of them is given weight 1. In other words $y[n] = 1 \cdot x[n] + 1 \cdot x[n-1] + 1 \cdot x[n-2] + 1 \cdot x[n-3]$. The filter now weights the last four savings and is therefore said to have length 4. The impulse response of the filter is the sequence of weights starting with the latest i.e. $[1 \quad 1 \quad 1 \quad 1]$. This type of filter has a bounded memory and weighs only a limited amount of samples; the filter is therefore called a *finite impulse response* (FIR) filter.

Let us instead assume you want your filter/system to compute your balance. A simple way to do that is to update the balance at each saving, i.e. the new balance is the old plus the saving amount. The difference equation for this system is $y[n] = 1 \cdot y[n-1] + 1 \cdot x[n]$. It is important that not only the input signal is part of the computation, but also old output signals. This makes the computation of the impulse response more involved since it is, per definition, a rule for how to weigh input signals in order to compute a certain output signal. If old output signals are part of the difference equation the system is called recursive. To solve the problem we see that the same equation can be used for old output signals, e.g. $y[n-1] = 1 \cdot y[n-2] + 1 \cdot x[n-1]$ and $y[n-2] = 1 \cdot y[n-3] + 1 \cdot x[n-2]$ etc. The important insight in this situation is that the output signal depends on all input samples (savings) up until now. The impulse response in this case is the infinitely long sequence $[1 \quad 1 \quad 1 \quad 1 \quad \cdots]$ and is called an *infinite impulse response* (IIR) filter.

In this simple example we have only used weights equal to 1, but with different weights for input signals of different age, the filter structure turns out to be very flexible. Each input signal can be divided into its frequency components, and in the rest of this course we will see how the filters can be used to enhance, attenuate, or delay different parts of these frequencies. In fact, it is exactly the same thing to combine input signals from different time instances to an output signal as it is to amplify, attenuate and delay the frequency components of a signal.

Overview

The detailed theoretical understanding of the different blocks in the system above will grow during the course. The computer exercises has three aims:

1. to illustrate applications of the theoretical concepts,
2. to describe how complicated systems can be built based on the basic components, and
3. to give examples of real-world signals from application areas where digital signal processing is used.

Development in signal processing is often done in the Matlab environment. Therefore the computer exercises also aim at giving experience with Matlab and to teach its different signal processing tools and how to use them. The step from development in Matlab to a real-time algorithm in C, which can be run on a microprocessor or a digital signal processor, is treated in later courses.

Part I

Laboration: A System for Recording

1 Introduction

The next system we will take a look at is shown in figure 1 and describes how a measured analog signal is converted to digital data which is then processed for the purpose of e.g. information extraction. This system is also very common and is used e.g. for recording of sounds for digital storage, to record other types of signals (for instance medical, electrical or mechanical) for analysis, and receiving transmitted data (if the microphone is replaced by an antenna). We call this system a recording/analysis/receiving unit.

An analog signal is measured with aid of a microphone, electrode, antenna or transducer of any kind. The signal has a frequency content up to a certain limit. The sampling theorem says that we must sample the signal (A/D convert) at a frequency which is twice as large as the largest signal frequency in order to get a correct discrete-time description of the analog signal. If we can not or do not want to sample at a such a high rate we can sample at a lower rate, provided the interesting information is in the lower frequency bands, but one must then eliminate frequencies above half the sampling frequency, in order to avoid aliasing and destruction of the signal. This is accomplished with the anti-aliasing filter. Usually noise is received at measurements, and the noise may reside at high frequencies, which is why anti-aliasing filters are almost always used before A/D conversion. After this the signal is read during a sampling period in the S/H circuit so that the A/D converter gets time to read the value of each sample.

In this exercise you will increase your understanding about a signal's frequency representation and how the DFT tool can be used. The theory is illustrated with an ECG signal. Your task is to analyze it in Matlab and extract the meaning of the different frequencies. The analog (electrical) signal was sampled (recorded) at the Department of Cardiology, Lund University Hospital, and stored digitally. Digital signal processing is performed off line by a computer. The analysis can also be made in real-time in order to use the system as an alarm.

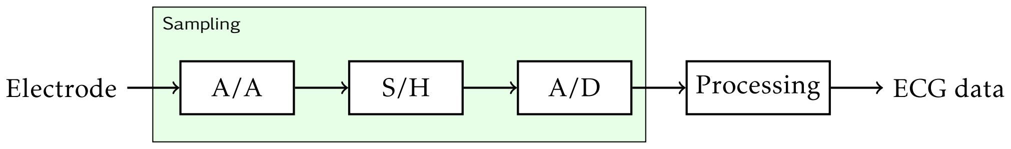


Figure 1: Sampling system.

2 An ECG Signal

The recorded ECG signal is sampled at 1 kHz. Thus, all frequencies over 500 Hz have been eliminated. This is not a problem since the body is not expected to produce faster variations than representable by approximately 250 Hz. Together with this lab is the file `ekg1.mat`.

Exercise 1 Download the ECG file to your Matlab directory. Load it into the Matlab workspace by writing:

```
load ekg1
```

Investigate the downloaded file:

```
size(ekg1)
```

You find out that the file `ekg1` consists of a discrete-time signal of 10 000 samples that are stored as a row vector (1 row and 10 000 columns). Using the sampling frequency 1 kHz, this means a length of 10 s. Investigate the file further by writing:

```
plot(ekg1)
```

You see a ECG file of 10 s containing 10 heart beats, where the peaks correspond to the electrical impulse that contracts the heart muscle and squeezes the blood out to the body, and the following small peaks correspond return to the original configuration. You also see a slow movement in the base line, i.e. the amplitude level between the beats. This variation is caused by breathing, and also by ground level variation due to the measurement process at the body. The Y-axis unit is micro volts. ....

Exercise 2 As mentioned the signal contains no information a bout how quickly it has been registered. Matlab uses sampling enumeration as X-axis for plots, as in the last exercise. It is always more interesting to have a time units on the X-axis, i.e. to map the sample enumeration to time units. This can be accomplished using the sampling frequency:

```
FT=1000;
N=10000;
n=0:N-1;
t=n/FT;
```

where `FT` is the sampling frequency, `N` is the signal length, `n` is the sample index counted from zero, and `t` is a new time indexation for the 10 000 samples that is given by the total number of samples divided by the number of samples per second. Plot the signal again but now supply the the X-axis unit:

```
plot(t,ekg1)
xlabel('time (s)')
ylabel('amplitude (microV)')
```

It can be seen that the pulse seem to be approximately 1 beat per second or 60 beats per minute. ....

Exercise 3 Use the DFT tool to see what frequencies the signal contains; the heart beat frequency should be the strongest. In Matlab the DFT is computed using the command `fft`. This command needs to know the number of evenly spaced frequencies in the interval from 0 to 1 to compute the spectrum. If you for instance choose to compute FFT for M points, the FFT will computed the spectrum at the normalized frequencies $[\frac{0}{M} \quad \frac{0}{M} \quad \cdots \quad \frac{M-1}{M}]$ If you want to know the physical frequencies this correspond to you must multiply them with the sampling frequency. We choose to compute the FFT at $M = 10\,000$ points.

```
M=10000;
ekgspek=fft(ekg1,M);
```

The vector `ekgspek` contain both the amplitude and the phase function and is therefore complex valued. We are here interested only in which frequencies are in the signal, which is why we plot only the amplitude function which corresponds to the absolute value of the complex spectrum.

```
plot(abs(ekgspek))
```

Since we did not supply any X-axis data the spectrum was plotted only against the enumeration of its samples. This information is difficult to interpret, but we do know the normalized frequencies they correspond to, and since we know the sampling frequency we can transform the normalized frequencies to the physical frequencies.

```
f=(0:M-1)/M*FT;
plot(f,abs(ekgspek))
xlabel('frequency (Hz)')
```

An important observation: Frequencies above half the sampling frequency (500 Hz) are only a mirrored image of the frequencies below 500 Hz. It is hence below 500 Hz we are to study the figure. We zoom in at the left part (0 Hz to 500 Hz in frequency and 0 to 800 000 in amplitude):

```
axis([0 500 0 800000])
```

It seems this signal is over-sampled, i.e. the prerequisites for the sampling theorem were filled with a great margin. Not much energy is above 50 Hz in the spectrum. We zoom further, this time in both variables:

```
axis([0 20 0 150000])
```

We see the frequencies constituting the ECG signal, meaning that the signal mathematically can be broken down to these frequencies, not that the signal has been created by addition of a number of sinusoids with these frequencies. This is of course not the case; the signal has been created by many cells simultaneously changing their surface potential which give rise to an electrical field in the chest, measured with an electrode. We have now designed a computer interface to the physicians, and we would like to interpret the spectral plot. We see that the frequency zero has the largest energy. The reason is the varying base line which albeit not varying with a large amplitude compared to the beats, but yet is spread all over the signal. This spectral peak we are not interested in. The next top is almost exactly at 1 Hz. This frequency with harmonics at 2 Hz, 3 Hz and 4 Hz, and so on, represents the peak waveform of the heart beats. The fundamental, i.e. the 1 Hz tone, corresponds to the repetition frequency of the signal. This frequency is thus a measure of the heart beat frequency. The computer program should therefore look for the first peak, overlooking the possible peak at 0 Hz. . . . .

Exercise 4 Finally we shall investigate the way a simple filter may affect the ECG signal. We start by plotting the signal but this time at the upper half of a figure.

```
subplot(2,1,1)
plot(t,ekg1)
```

We now try to decrease the noise of this signal by specifying that the output signal at time n is a weighted average of the fifteen last samples. The filter is then fifteen coefficients with equal values, set for example to 0.2, for the input samples with index n to $n-14$. The filtering is done using convolution (the `conv` function in Matlab) of the input signal and the filter coefficients.

```
h=0.2*ones(1,15);
y=conv(ekg1,h);
y=y(15:end);
```

The length of the convolution is the input signal length plus the filter length minus one. The last row takes away the transient at the beginning (the filter doesn't have data at all its coefficients at the beginning), and makes the length of the output signal equal the length of the input signal.

```
subplot(2,1,2)
plot(t,y)
```

We see a signal of considerably smaller noise. The signal is so clean that we can see a small peak before each of the great peaks. The small peaks correspond to atrial contraction which fill the ventricles with blood, so they are full when the great contraction takes place at the great peak. . . . .

Part II

Laboration: Delay Processor

1 Introduction

In this lab you will work with a real time application in Matlab. Read the part titled "Configuring the Real Time Environment" at the end of this document for information on how to connect and configure the microphones and the speaker in order to proceed with this lab.

2 Preparation Tasks

Preparation task 1 In this lab, we will look at delay and echo filters. Given a filter with the impulse response $h(n)$

$$h(n) = \frac{1}{3}\begin{bmatrix}1 & 0 & 0 & 0 & 1\end{bmatrix} \tag{1}$$

determine its Fourier transform $H(\omega)$

$$H(\omega) = \sum_k h(k)e^{-j\omega k} \tag{2}$$

of the filter. ....

Preparation task 2 Sketch the frequency response of the filter $h(n)$ from preparation task 1.

3 Tasks

Exercise 1 Determine the delay in a GSM channel by calculating the cross-correlation between a signal before and after passing through the channel. Download the file `gsmsig.mat`, load it into Matlab and listen to it.

```
load gsmsig;
N = size(gsmsig, 1);
t = (0:N-1)/10000;
plot(t, gsmsig);
```

Listen to the two channels at the same time (the before and the after channels are played back in the left and the right channel, respectively):

```
soundsc(gsmsig, 10000);
```

Determine the delay between the two signals with the help of the Matlab function `xcorr`. What is the delay of the GSM channel? ....

Exercise 2 We start by looking at a simple echo in Matlab. To produce an echo effect, we use the system presented in figure 1 and assuming that $\alpha = 1$. First create an input signal to the system by generating a 2 s long sweeping sinusoid from 300 Hz to 800 Hz sampled at 10 000 Hz.

```
t = (0:2*10000-1)/10000;
x = chirp(t, 300, 2, 800);
soundsc(x, 10000);
```

Generate an echo signal by padding two different copies of the signal at the beginning and at the end, respectively, with zeros. The signal padded at the beginning is the delayed signal, and the amount of padding determines the delay D. Add the two padded signals and weigh the delayed signal by the scale factor β. In this example, $D = 500$ and $\beta = 0.9$.

```
z = zeros(1, 500);
x1 = [x, z];
x2 = [z, x];
y = x1 + 0.9*x2;
soundsc(y, 10000);
```

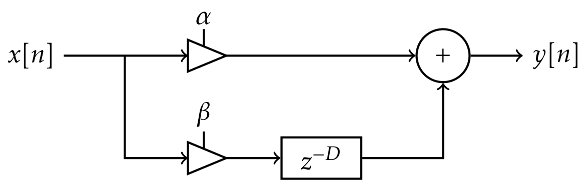


Figure 1: First-order echo processor with variable delay and gain.

Listen to the resulting signal. Experiment with different delay D and scaling α. Also try making an echo signal from one of the two GSM channel signals from task 1 ....

Exercise 3 The echo processor delays the signal by D samples and multiplies the signal with the damping factor a where $|a| < 1$. Start the echo processor in Matlab with the command:

```
    reverb
```

and select the *First order (FIR)* filter option. The echo processor is shown in figure 1 Start the processing with the *Start* button and stop processing with the *Stop* button. Talk in the microphone, adjust the delay and gain parameters and listen to the processed output signal. Try to estimate the value of the delay when it is difficult to talk. ....

Exercise 4 Write the difference equation for $y(n)$ in task 3. Determine the poles and zeros and make a pole-zero plot. Does the system have linear phase? ....

Exercise 5 Add some more delay lines to the echo processor as shown in figure 2 by selecting the *Third order (FIR)* filter option. Repeat tasks 3 and 4 with this echo processor. ......

Exercise 6 Add an infinite number of delay lines to the echo processor by feeding back the signal as shown in figure 3 by selecting the *Recursive (IIR)* filter option. Repeat tasks 3 and 4 with this echo processor. ....

Exercise 7 A choir effect can be created from a single audio channel with an echo processor whose delay is time-varying. An illusion of a stereo effect can also be made with two separate echo processors with slightly different parameters. When recording a choir with two microphones, each microphone records different mixtures of the choir, but also a cross-mixture of the other microphone. This is simulated by a final summation stage where the two choir signals are cross-mixed.

Two echo processors, generating the left and right audio channels, and a cross-mixer comprise the choir processor shown in figure 4. Each echo processor can be configured independently by the the base delay d, the relative swing amplitude a and the swing frequency f. The delay of an echo processor is thus time dependent, and is modeled as $D(t) = d + d \cdot a \cdot \sin(2\pi \cdot f \cdot t)$. The two terms are denoted the base delay and the delay modulation.

Start the chorus processor.

```
    chorus
```

The default parameter set is a pass-through on the left channels without any chorus effect. Increase the parameters for the different echo processors to see what happens to the output signal, and adjust the cross-gain to mix the two channels. Try to find a set of parameters that work well for your voice. A good choir effect, however, is often achieved when the echo processors have slightly different base delays that gives the appearance of multiple different echo paths, and when the delay modulation is not too strong and different for all the echo processors. ....

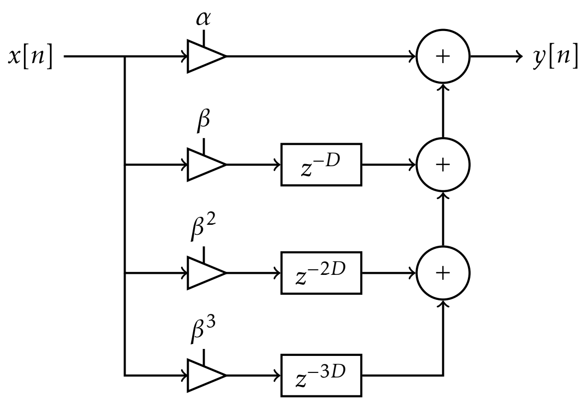


Figure 2: Third-order echo processor with variable delay and gain.

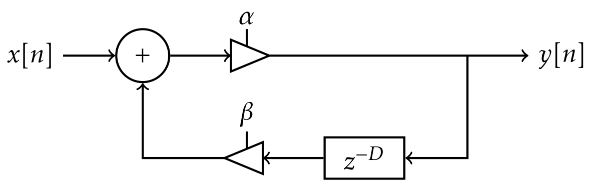


Figure 3: Recursive echo processor with variable delay and gain.

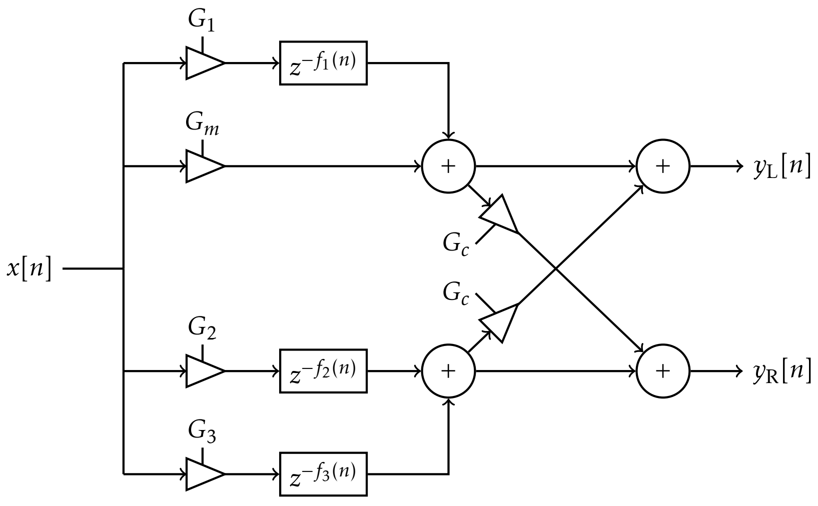


Figure 4: Chorus processor with multiple echo processors.

Part III

Laboration: IIR Filter Design

1 Introduction

A time-discrete system $H(z)$ can be described by the location of its poles and zeros in the z-plane. From the pole-zero plot, we can estimate the frequency response $H(\omega)$ by looking at the value of $H(z)$ on the unit circle. In general, the value of $|H(\omega)|$ is high when close to a pole, and low when close to a zero.

2 Preparation Tasks

Preparation task 1 Match the pole-zero plots with their respective amplitude response function in figure 1. ....

Preparation task 2 Sketch the approximate amplitude response functions of the pole-zero plots in figure 2. ....

3 Laboration Tasks

The program `mkiir` allows you to visually design an IIR filter by placing poles and zeros in the z-plane. Read the part titled "IIR Filter Design Application" at the end of this document for information on how to use the program.

Exercise 1 Using the filter design program, place a single pole or zero and move it around on around the z-plane (note that a single pole or zero means the *pair* of complex conjugated poles or zeros that automatically appear in the pole-zero plot). What happens to the frequency response as you move the pole or the zero around the z-plane? What is the relationship between the frequency response and the angle and radius of the pole or zero? ....

Exercise 2 Select the prototype filter *Low pass* and a filter specification appears in the frequency response window. Place poles and zeros in the z-plane and try to fulfill the filter specification by having the amplitude response inside the unshaded region. Use as few poles and zeros as possible. Don't forget to use the gain parameter to shift the amplitude response vertically to fit within the design constraints. ....

Optional Select another prototype filter and try to fulfill another design. Try to design the high-pass filter or the band-pass filter. ....

Optional You can also use Matlab to design a filter. The following code designs a Chebychev band-pass filter with stop-band ripple.

```
[b, a] = cheby2(4, 40, [.15, .35]*2);
```

The filter is a 4:th order IIR filter, the stop-band ripple (stop-band attenuation) is 40 dB, and the stop-band is between the normalized frequencies 0.15 and 0.35. Save the filter:

```
save('c:\path\myfilter.mat', 'b', 'a');
```

where `c:\path\` is some path where you can save your files and scripts. Load it into the IIR filter design program to see where the poles and zeros are located.

Note: The factor 2 on the frequency range in `cheby2` is due to Matlab having different notation of normalized frequency, where 1 corresponds to half the sampling frequency instead of the whole sampling frequency. ....

The filters you have designed thus far have been described by stop-bands and pass-bands. Sometimes it is necessary to not block a range of frequencies but rather a single frequency. You can design a filter blocking a single frequency by placing a zero on the unit circle at the desired frequency.

One major drawback with placing only a single zero, corresponding to an FIR-filter where all the poles are located at the origin of the z-plane, is that a large area around the zero is significantly attenuated. To achieve a near flat response except close to the frequency we wish to cancel, the poles can be moved out from the origin towards the zero. The closer the poles are to the zeros, the narrower the notch. These filters are called *notch filters*. The equation for a single-frequency notch filter is

$$H(z) = \frac{\left(1 - e^{j\omega_0} z^{-1}\right)\left(1 - e^{-j\omega_0} z^{-1}\right)}{\left(1 - r \cdot e^{j\omega_0} z^{-1}\right)\left(1 - r \cdot e^{-j\omega_0} z^{-1}\right)} \tag{3}$$

where ω_0 is the normalized frequency to be canceled and r is the radius of the pole. The two numerator terms are the two complex conjugate zeros, and the two denominator terms are the two complex conjugate poles. Filters with multiple notches is designed by multiplying more complex conjugate poles and zeros at the desired frequencies to the numerator and denominator. The radius r controls the width of the notch, and is typically close to and less than 1 so that is is placed just inside the unit circle. The zeros are placed on the same frequencies, but *on* the unit circle.

Exercise 3 Use the program `jukebox` to load a small sequence of a music signal into Matlab. Pick any signal from the list.

```
[x, fs] = jukebox;
```

The music files have been disturbed by sinusoidal signals with unknown frequencies. Plot the power spectral density of the signal and examine the frequency contents.

```
pwelch(x, [], [], [], fs);
```

Also listen to the signal.

```
sound(x, fs);
```

The graph that appears will have a few narrow peaks indicating some disturbing sinusoidal signals. Which frequencies are disturbing the signal you selected? ....

Exercise 4 From the list of frequencies you wish to cancel, you can create a list of the poles (located near the unit circle at radius `r`) and the zeros (located on the unit circle):

```
z = exp(-1j*w*pi*freq/fs);
p = exp(-1j*w*pi*freq/fs)*r;
```

The vector `freq` lists all the frequencies you wish to cancel. Don't forget that the frequency spectrum is symmetric, and the list of frequencies has to include the negative frequencies as well. The numerator and the denominator polynomials of the filter in (3) can now be created:

```
b = poly(z);
a = poly(p);
```

Filter your signal and plot the power spectral density:

```
y = filter(b, a, x);
pwelch(y, [], [], [], fs);
```

If the sinusoidal disturbances are not canceled, go back and redesign the filter with new or adjusted notch frequencies. If the disturbances have been removed, listen to the filtered signal:

```
sound(y, fs);
```

The disturbances should be completely removed. ....

Optional Save your notch filter and import it into the `mkiir` program.

```
save('c:\path\mynotch.mat', 'b', 'a');
```

Do the location of the poles and zeros match your expectations? ....

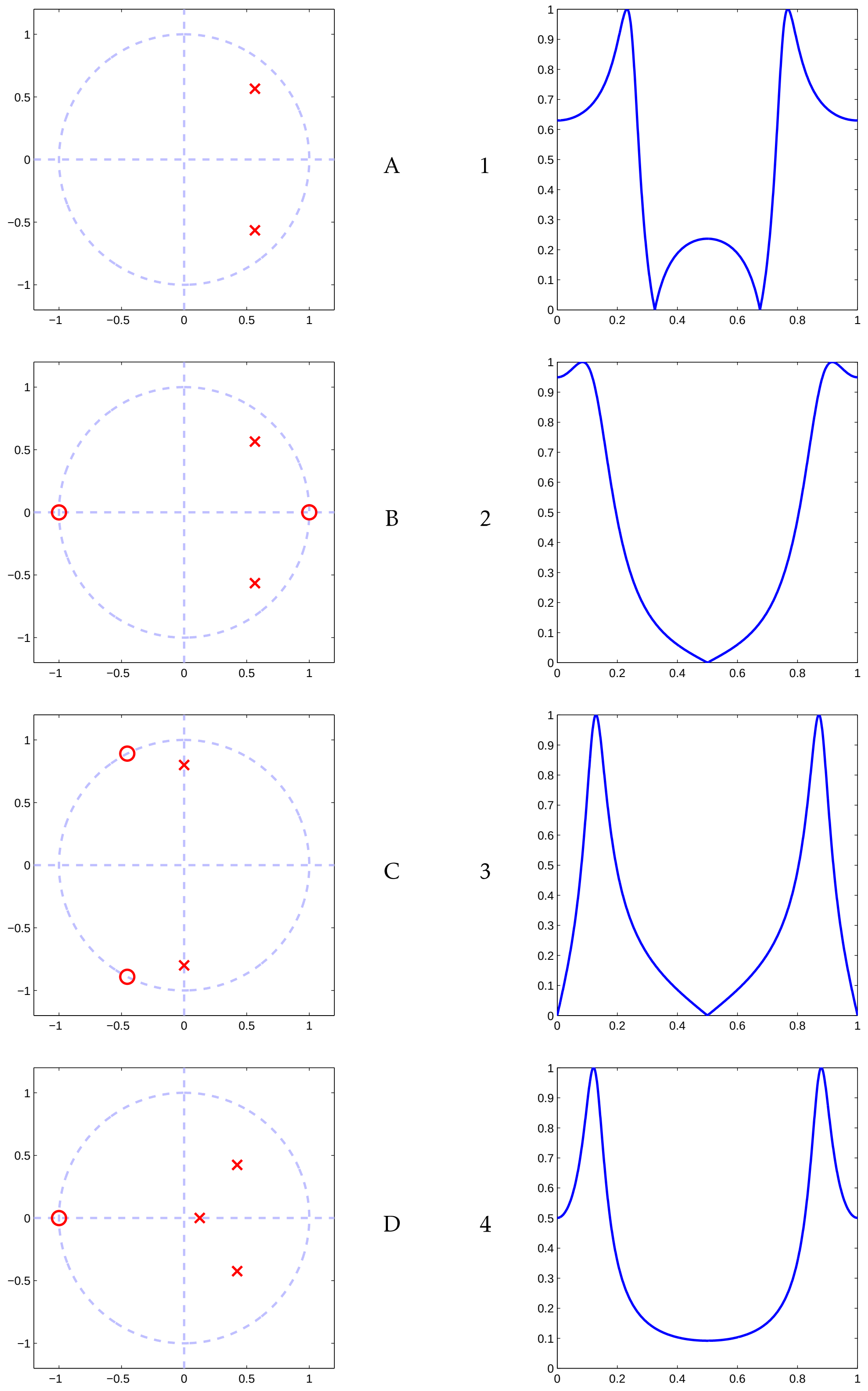


Figure 1: Match pole-zero plot with amplitude response.

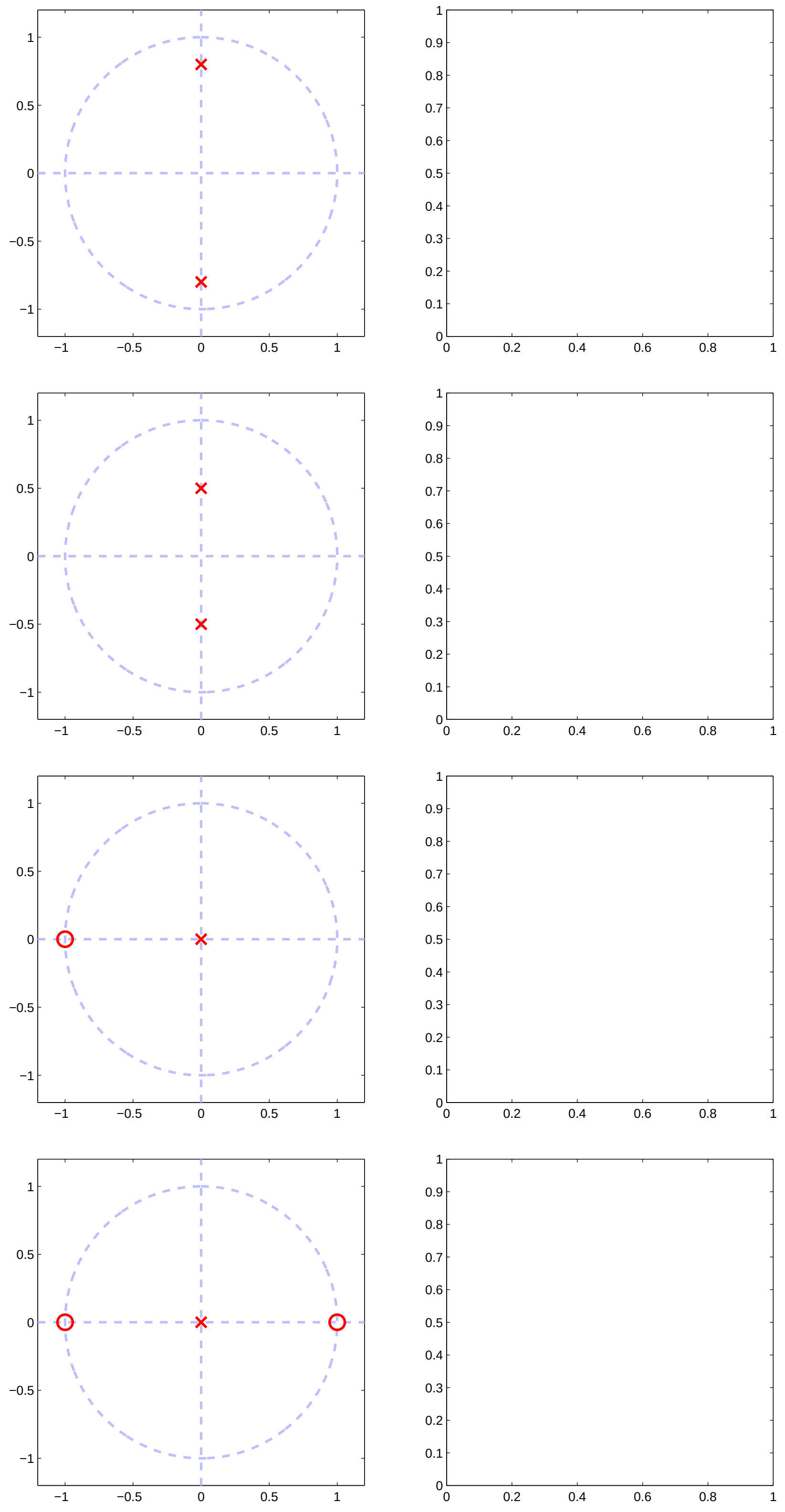


Figure 2: Sketch amplitude response.

Part IV

Laboration: Image Filtering

1 Introduction

This laboration aims to give an understanding about the concept of *discrete frequency*. Both one-dimensional time signals and two-dimensional images will be used as to illustrate the concept. The discrete Fourier transform will be introduced and the concept of frequency response will be demonstrated. Additionally, the connection between frequency content and visual appearance will be introduced.

2 Filtering of One-dimensional Signals

Exercise 1 The following code creates a square pulse signal and a moving average filter, and then filters the signal with the filter. Start by creating the signal.

```
N = 256;
n = (0:N-1);
f = 12/N;                % normalized frequency
x = square(2*pi*n*f);    % square pulse signal with frequency f
```

Next, create the filter and apply it to the signal.

```
L = 5;
h = 1/L * ones(L, 1);    % 5-tap moving average filter
y = filter(h, 1, x);
```

Finally, plot the result to compare the input signal to the moving average filter and the output signal from the filter.

```
subplot(2, 1, 1); plot(n, x, '.-');
subplot(2, 1, 2); plot(n, y, '.-');
```

Try different frequencies of the square signal and see what happens to the filtered signal. .

Exercise 2 The filter in task 1 is a moving average filter with the filter vector defined as:

$$h(n) = \frac{1}{7} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{4}$$

Repeat task 1 but with the following filter vector:

$$g(n) = \delta(n) - h(n) \tag{5}$$
$$= \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} - \frac{1}{7} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{6}$$
$$= \frac{1}{7} \cdot \begin{bmatrix} -1 & -1 & -1 & 6 & -1 & -1 & -1 \end{bmatrix} \tag{7}$$

Explore the differences between the filtered signal when using the filters $h(n)$ and $g(n)$ at different frequencies of the input signal. What happens to the input signal at sharp edges, and what happens on areas with constant amplitude? ....

3 Filtering of Two-dimensional Signals

A filter operation can be described as a filter mask sliding over a signal. For every sample of the input signal, the filter mask is placed on the current sample and its neighbouring samples. The signal and the mask is multiplied, and the sum of all products is the filtered output sample value for the current input sample. The same idea of a filter mask is applies to two-dimensional signals as well, for example images.

Exercise 3 Download the file `lena.tif`. The following code loads and displays the image:

```
I = imread('lena.tif');
x = double(I)/255;
figure;
imshow(x);
```

Apply a two-dimensional 5×5 moving average filter:

```
L = 5;
h = 1/(L*L) * ones(L);
y = filter2(h, x);
figure;
imshow(y);
```

Experiment with different filter sizes and observe the effects. . . . .

Exercise 4 The filter in task 3 is a two-dimensional moving average filter with the filter vector defined as:

$$h(m,n) = \frac{1}{5^2} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{8}$$

Repeat task 3 but with the following filter vector:

$$g(m,n) = \delta(n,n) - h(n,n) \tag{9}$$

$$= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} - \frac{1}{5^2} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{10}$$

$$= \frac{1}{5^2} \cdot \begin{bmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & 24 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{bmatrix} \tag{11}$$

Explore the differences between the filtered image when using the filters $h(m,n)$ and $g(m,n)$ at different frequencies of the input signal. What happens to the input signal at sharp edges, and what happens on areas with constant amplitude? . . . .

Appendix A

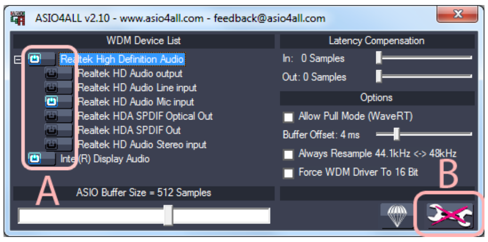
Configuring the Real Time Environment

1 Introduction

This section describes how to setup the real time application framework on the lab computer. All realtime applications run from Matlab in these labs start and stop the realtime processing with *Start* and *Stop* buttons. Any parameter that can be changed when processing has started has immediate effect when changed.

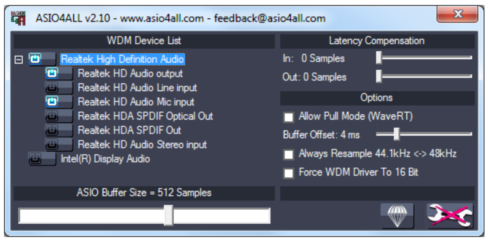
2 Configuration

Audio Channels Start the configuration script from Matlab with `configure panel`. Enable and disable the audio sources as shown in either of the two images below. Select the audio channels (A) according to the desired setup: left image for playback through the speakers built into the monitor, and right image for playback through the headphones connected to the computer. Press the wrench icon (B) if necessary to enable the detailed audio configuration options as shown in the image.



Realtek High Definition Audio
Realtek HD Audio output
Realtek HD Audio Line input
Realtek HD Audio Mic input
Realtek HDA SPDIF Optical Out
Realtek HDA SPDIF Out
Realtek HD Audio Stereo input
Intel(R) Display Audio

In: 0 Samples
Out: 0 Samples
Allow Pull Mode (WaveRT)
Buffer Offset: 4 ms
Always Resample 44.1kHz <-> 48kHz
Force WDM Driver To 16 Bit
ASIO Buffer Size = 512 Samples



Realtek High Definition Audio
Realtek HD Audio output
Realtek HD Audio Line input
Realtek HD Audio Mic input
Realtek HDA SPDIF Optical Out
Realtek HDA SPDIF Out
Realtek HD Audio Stereo input
Intel(R) Display Audio

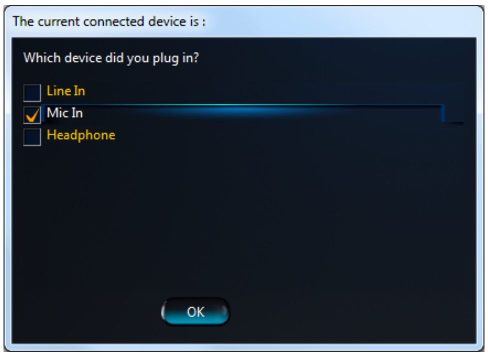
In: 0 Samples
Out: 0 Samples
Allow Pull Mode (WaveRT)
Buffer Offset: 4 ms
Always Resample 44.1kHz <-> 48kHz
Force WDM Driver To 16 Bit
ASIO Buffer Size = 512 Samples

Verify the setup from Matlab with `configure show`. The left column below shows the correct audio channel list for playback through the speakers, and the right column shows the correct audio channel list for playback through the headphones. Ensure that the *Mic input* and not the *Line input* input channels are selected, and that either the *Display audio* or the *Audio output* output channels are selected.

```
>> configure panel
>> configure show
In:
    'HD Audio Mic input 1'
    'HD Audio Mic input 2'

Out:
    'Display Audio Output 1 1'
    'Display Audio Output 1 2'
```

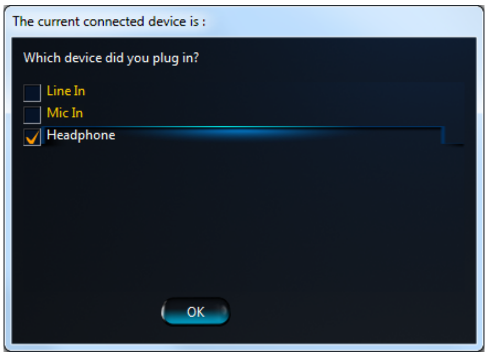
```
>> configure panel
>> configure show
In:
    'HD Audio Mic input 1'
    'HD Audio Mic input 2'

Out:
    'HD Audio output 1'
    'HD Audio output 2'
    'HD Audio output 3'
    'HD Audio output 4'
    'HD Audio output 5'
    'HD Audio output 6'
    'HD Audio output 7'
    'HD Audio output 8'
```

Microphone and Headphones Connect the microphone, and optionally the headphones, in either the front panel or the back panel on the computer. Select the proper item type you connected when requested.


The current connected device is :

Which device did you plug in?

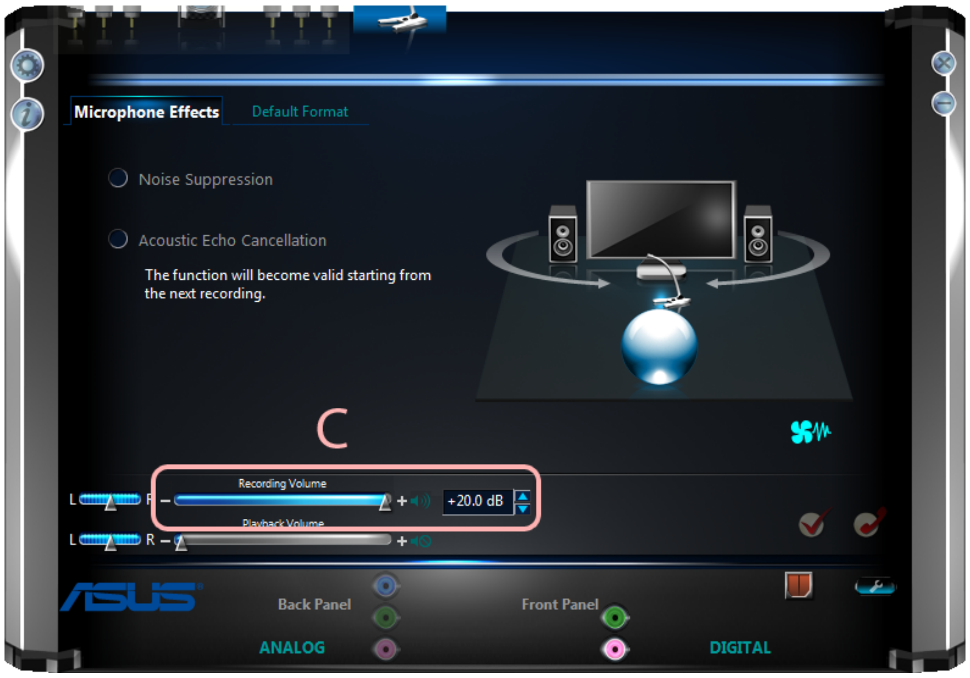
- [ ] Line In
- [✓] Mic In
- [ ] Headphone


The current connected device is :

Which device did you plug in?

- [ ] Line In
- [ ] Mic In
- [✓] Headphone

Volume When connecting the microphone, ensure that the recording volume and gain (C) is turned up sufficiently. Set to maximum recording volume and 20 dB to 30 dB gain.



The function will become valid starting from the next recording.

Recording Volume: +20.0 dB

Appendix B

IIR Filter Design Application

1 IIR Filter Design Application

The program `mkiir`, shown in figure 4, allows you to visually design an IIR filter by placing poles and zeros in the z-plane. The program shows a complex pole-zero plane (top) and the filter amplitude response (bottom) on the left hand side, and program options on the right hand side.

2 Pole-Zero Plot

The pole-zero plot shows the unit circle and the real and imaginary axes. This area is where you place, move and delete your poles and zeros. Poles and zeros are automatically placed in complex conjugate pairs. Therefore, only IIR filters with odd-ordered numerator and denominator polynomials can be designed by and imported to the program.

3 Amplitude Response

The amplitude response updates continuously with the current filter response as you add, move or delete poles or zeros from the filter. This area can also display a prototype filter specification you can attempt to design using poles and zeros.

4 Program Options

The settings panel on the right hand side allows you to control the behaviour and actions of the filter design application.

Modify Poles or Zeros

Select whether to add, move and delete poles or zeros. Poles or zeros, depending on this option, are added to the filter by left clicking on the pole-zero plot. Poles or zeros are moved by left clicking and dragging an existing pole or zero, and are removed by left clicking near an existing pole or zero.

Poles and zeros can only be placed on or inside the unit circle on the z-plane. Poles or zeros placed outside the unit circle will be projected onto the unit circle so that you can place a point exactly on the unit circle.

Filter Prototype

Show a filter prototype in the amplitude response area. Selecting a filter prototype overlays a pre-defined design constraint on the amplitude response for you to design a filter against.

Filter Gain

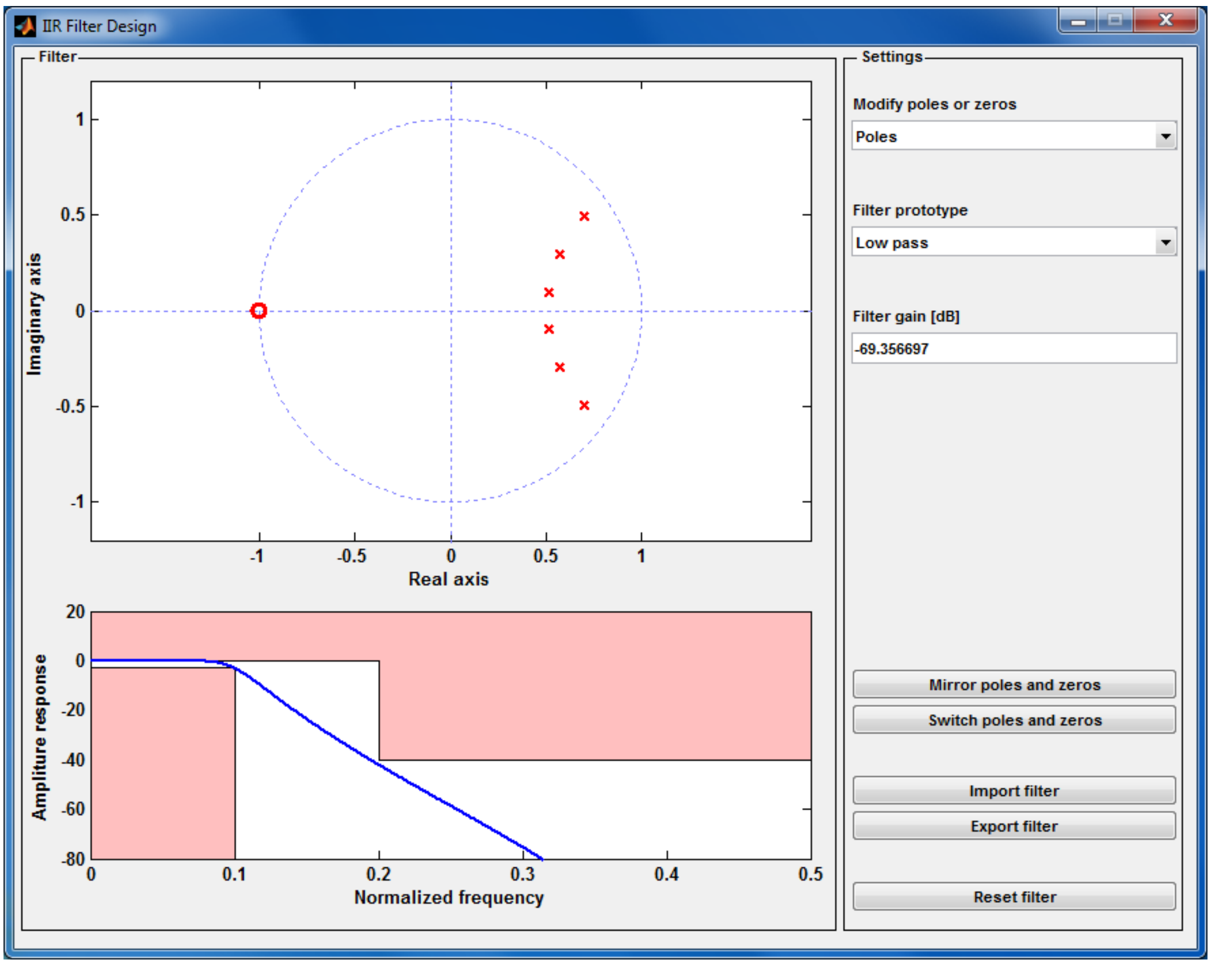
Set a global filter gain or attenuation to vertically shift the amplitude response. This option allows you to scale the frequency response to fit within the design constraint.

Mirror and Switch Poles and Zeros

Mirror poles and zeros across the imaginary axis, or switch poles to zeros and zeros to poles.

Import and Export Filter

Import pre-designed filter coefficients from Matlab, or export your filter to Matlab. This allows you to design a filter in Matlab and import it into the design application to see how different filter designs places the poles and zeros, or to export your filter coefficients for use in Matlab.



Modify poles or zeros: Poles

Filter prototype: Low pass

Filter gain [dB]: -69.356697

Figure 1: Pole-zero based IIR filter design application.