LAB 5: Introduction to Simulink and Filter Design using MATLAB.

Objective

This lab:

- Gives you a deeper understanding of the analog and digital filter design techniques in MATLAB.
- Gives you a deeper understanding of the filter design techniques in MATLAB using the Filter Design & Analysis Tool (FDAT).
- Helps you to represent, play, construct and plot audio signals in MATLAB.
- Introduces you to Simulink.

Hint: Part of this lab uses Simulink, a companion to MATLAB. The lab is self-contained, in the sense that no additional documentation for Simulink is needed.

Introduction

- **Lowpass Filter**: The lowpass filter (LP-filter) is characterized by the attenuation of the higher frequencies and passing the low, sometimes with a gain. The phase characteristics of this filter depend on the order of the filter and the implementation [1].
- **Highpass filters**: The Highpass filter (HP-filter) is characterized by the attenuation of the low frequencies and passing the high, sometimes with a gain. The phase characteristics of this filter depend on the order of the filter and the implementation [1].
- **Bandpass Filter**: The bandpass filter selectively allows passage for all the frequencies of the certain range and rejects/attenuates frequencies out of that range, sometimes with a

gain. The phase characteristics of this filter depend on the order of the filter and the implementation [1].

- **Bandstop Filer**: The Bandstop filter selectively blocks the passage of frequencies within certain range, and allows free passage for frequencies outside the range. It is the opposite of a band-pass filter [1].
- **Notch Filter**: The notch filter is a bandstop filter with a narrow stop band. The notch filter is characterized by containing one or more sharp attenuation slope in its frequency response. This filter type is generally used to remove a disturbance of a known narrow band frequency. The width of the attenuation notch and the maximum attenuation depends on the order and implantation of the filter. The maximum gradient of the phase slope is at the center of the attenuation frequency [1].

Signal Processing Tool

The Signal Processing Toolbox application, **`SPTool`**, provides a rich graphical environment for signal viewing, filter design, and spectral analysis [2]. You can use **`SPTool`** to analyze signals, design filters, analyze filters, filter signals, and analyze signal spectra. You can accomplish these tasks using four GUIs that you can access from within **`SPTool`**:

- **The Signal Browser** is for analyzing signals. You can also play portions of signals using your computer's audio hardware (sound card). Using the Signal Browser you can
    - View and compare vector/array signals.
    - Zoom in on a range of signal data to examine it more closely.
    - Measure a variety of characteristics of signal data.
    - Play signal data on the audio hardware.

To open/activate the Signal Browser for the **`SPTool`**, Click one or more signals (use the Shift key for multiple selections) in the Signals list of **`SPTool`**. Then Click the View button in the Signals list of **`SPTool`**.

- **The Filter Designer** is for designing or editing FIR and IIR digital filters. Note that the **`FDATool`** is the preferred GUI to use for filter designs [3]. **`FDATool`** is discussed later in this lab.

- **The Filter Viewer** is for analyzing filter characteristics.
- **The Spectrum Viewer** is for spectral analysis.

Open **`SPTool`** by typing **`sptool`** at the command prompt >>.

```
>>sptool
```

You see 3 panes - Signals, Filters and Spectra. View all the signals in these panes. You can play the signals in Signals through using the speaker icon in the Signals browser. (Note: The speaker will only play the signal between the two cursors on the Signal Browser). When you are trying to view spectra, note that many methods of determining spectra, including FFT are available. At this point, just use the FFT. In the Filters pane just View all three filters (LSip, PZip, FIRbp) at this time and get a sense of the capabilities of Filters.

Filter Design and Implementation

The design of a digital filter is carried out in three steps:

- **Specifications:** Before we can design a filter, we must have some specifications. These specifications are determined by the applications.
- **Approximations:** Once the specifications are defined, we use various concepts and mathematics to come up with a filter description that approximates the given set of specifications.
- **Implementation:** The product of the above step is a filter description in the form of either a difference equation, or a system function $H(z)$, or an impulse response $h(n)$. From this description we implement the filter in hardware or through software on a computer.

The goal of filtering is to perform frequency-dependent alteration of a signal. A simple design specification for a filter might be to remove noise above a certain cutoff frequency. A complete specification determines the amount of *passband ripple* (R_p, in decibels), *stopband attenuation* (R_s, in decibels), or *transition width* ($W_s - W_p$, in hertz).

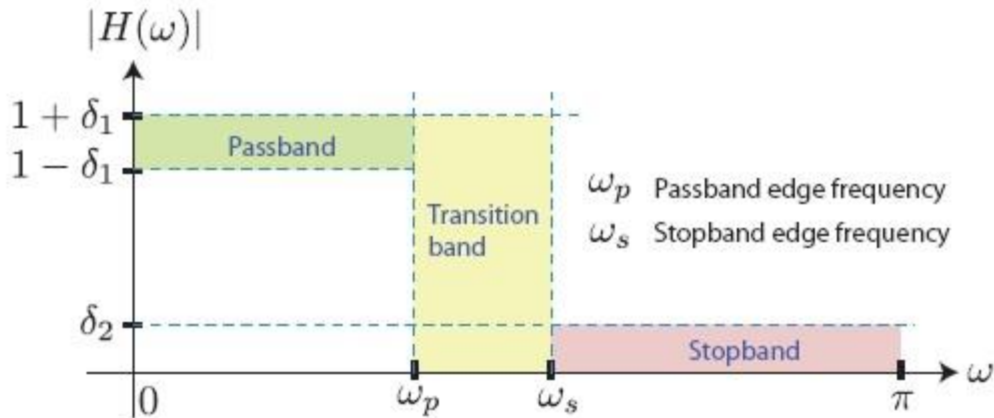


Figure 1. Specifications for a realizable filter

Analog Filter Design Using MATLAB

MATLAB provides many filter design tools [3][4]. Most of the tools are aimed at *digital filter design*, but some of the tools also support *analog filter design*. We briefly consider some analog filter design tools. Here we briefly summarize the characteristics of the lowpass version the most widely used analog filters in practice: *Butterworth*, *Chebyshev* (Type-1), and *Elliptic*.

- **Butterworth Filter**: This filter is characterized by the property that its magnitude response is flat in both *passband* and *stopband*. The magnitude squared response of *N*-th order lowpass filter is given by

$$|H_a(j\Omega|^2 = \frac{1}{1+\left(\frac{\Omega}{\Omega_c}\right)^{2N}}$$

where *N* is the order of filter and Ω_c is the cutoff frequency in ***rad/sec***. To design an analog Butterworth filter using MATLAB, one uses the command

```
[b, a] = butter (N, cutoff_freq,'s')
```

This command tells MATLAB to design a Butterworth filter of order N and *cutoff frequency* **`cutoff_freq`**. The **`'s'`** tells MATLAB to design an analog filter. (Without this command,

MATLAB designs a digital filter.) The vectors **`a`** and **`b`** hold the coefficients of the denominator and the numerator (respectively) of the filter's transfer function.

Example 1. Giving MATLAB the commands

```
>> [b, a] = butter (4, 100,'s');
>> g = tf(b, a)
```

causes MATLAB to respond with

```
g =

                          1e08
  ---------------------------------------------------
  s^4 + 261.3 s^3 + 3.414e04 s^2 + 2.613e06 s + 1e08

Continuous-time transfer function.
```

Giving MATLAB the command

```
>> bode(g,{30,4000}); grid;
```

causes MATLAB to respond with Figure 2. (The term {30,4000} in the **`bode`** command causes the command to plot the magnitude and phase response from **30 rad/s** out to **4,000 rad/s**.) . Two points are worth noting. Looking at the magnitude plot, one sees that at **100 rad/s** the response seems to have decreased by about **3 dB**- as it should. Also, one notes that from **100 rad/s** to **1000 rad/s** the response seems to drop by about **80 dB**. As this is a fourth order filter its should be **4 × 20 dB/dec**.

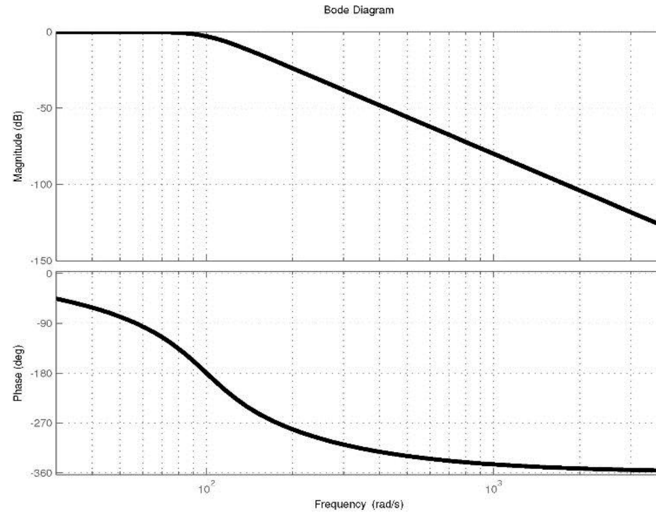


Figure 2. The magnitude and phase plot of the fourth-order low-pass Butterworth filter with a cutoff frequency of 100 rad/s.

- **Chebyshev Filter**: There are two types of Chebyshev filters. The Chebyshev-I filters have *equiripple response* in the **passband**, while the Chebyshev-II filters have *equiripple response* in the **stopband**. The magnitude-squared response of a Chebyshev-I filter is

$$|H_a(j\Omega|^2 = \frac{1}{1 + \varepsilon^2 T_N^2 \left(\frac{\Omega}{\Omega_c}\right)^{2N}}$$

where N is the order of the filter, ε is the passband ripple factor, which is related to R_p, and $T_N(x)$ is the Nth-order Chebyshev polynomial given by

$$T_N(x) = \begin{cases} \cos(N\cos^{-1}(x)), & 0 \le x < 1 \\ \cos(\cosh^{-1}(x)), & 1 < x < \infty \end{cases} \quad where \quad x = \frac{\Omega}{\Omega_c}$$

MATLAB provides functions called **`cheby1`** and **`cheby2`** to design Chebyshev- type I and Chebyshev- type II respectively. Read more about these functions (**`doc cheby1, doc cheby2`**).

- **Elliptic Filter**: These filters exhibit *equiripple* behavior in the **passband** as well as in the **stopband**. They are similar in magnitude response characteristics to the FIR equiripple filters. Therefore elliptic filters are optimum filters in that they achieve the minimum

order N for the given specifications. The magnitude-squared response of an elliptic filter is given by

$$|H_a(j\Omega|^2 = \frac{1}{1+\varepsilon^2 U_N^2\left(\frac{\Omega}{\Omega_c}\right)^{2N}}$$

where N is the order of the filter, ε is the **passband** ripple factor, which is related to R_p, and $U_N(.)$ is the Nth-order Jacobian elliptic function. MATLAB provides a function called **`ellip`** to design Elliptic filter.

Finite impulse response (FIR) and Infinite impulse response (IIR) Filters in MATLAB

In practice we would prefer either a rational system function corresponding to **FIR** or **IIR** filters. Thus we consider using the difference equation model given in (1) and (2) representing the descriptions in the time- and frequency-domain, respectively.

$$y(n) = -\sum_{k=1}^{N} a_k y(n-k) + \sum_{k=0}^{M} b_k x(n-k) \tag{1}$$

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1+\sum_{k=1}^{N} a_k z^{-k}} \tag{2}$$

In general, IIR filters are less complex than FIR filters (the difference being that for FIR filters there is a restriction that $a_k = 0$ all k), as they require fewer parameters and less memory for the same *"quality"* of filter performance. IIR filter design methods differ from FIR primarily in how performance is specified. For loosely specified requirements, a *Butterworth* filter is often sufficient. More rigorous filter requirements can be met with the *Chebyshev* and *elliptic* filters.

In addition, the primary advantage of IIR filters over FIR filters is that they typically meet a given set of specifications with a much lower filter order than a corresponding FIR filter. This has the obvious implementation advantages. IIR filters have nonlinear phase. However, if data processing is to be performed offline, then the effects of the nonlinear phase can be eliminated. So let us assume that the entire input data sequence is available prior to filtering. This allows for

a non-causal, zero-phase filtering approach (via the **`filtfilt`** function), which eliminates the nonlinear phase distortion of an IIR filter. (**`doc filtfilt`**)

In practice one wonders about which filter (FIR or IIR) should be chosen for a given application and which method should be used to design it. Because these design techniques involve different methodologies, it is difficult to compare them. One basis of comparison is the number of multiplications required to compute one output sample in the standard realization of these filters.

The Signal Processing Toolbox used functions to estimate the minimum filter order that meets a given set of filter specifications. Luckily, MATLAB has graphical user interface (GUI) filter design program, which requires us to fill a few fields and to click a few buttons.

Filter Configurations

First, recall that in digital signal processing (DSP) we are dealing with sampled signals, so we have to normalize the frequencies to the ***Nyquist frequency*** (defined as one-half the sampling frequency). All the filter design functions in the Signal Processing Toolbox operate with normalized frequencies, so that they do not require the system sampling rate as an extra input argument. The normalized frequency is always in the interval $0 \leq f \leq 1$. For example, with a 2000Hz sampling frequency, 400 Hz is $400/1000 = 0.4$. To convert normalized frequency to angular frequency around the unit circle, multiply by π. To convert normalized frequency back to Hertz, multiply by half the sample frequency. So we have the following:

- *Lowpass* filters remove high frequencies (near 1).
- *Highpass* filters remove low frequencies (near 0).
- *Bandpass* filters pass a specified range of frequencies between 0 & 1
- *Bandstop* filters remove a specified range of frequencies between 0 & 1

Filter Specifications in MATLAB

- ω_p - Passband cutoff frequencies (normalized)
- ω_s - Stopband cutoff frequencies (normalized)
- R_p - Passband ripple: deviation from maximum gain (dB) in the passband
- R_s - Stopband attenuation: deviation from 0 gain (dB) in the stopband

Recall the concept of the *ideal lowpass* filter, which is simply visualized in the frequency domain as the rectangle function (of a specified width and centered at the origin). Also recall that this filter, although handy in theory, is not implemented in practice. There are several reasons why ideal (lowpass, bandpass, highpass, and band-stop) filters are not used in real-life:

1. The impulse response, $h(n)$, is non-causal as a consequence of the Paley-Wiener theorem, which implies these filters cannot be implemented in practice on a DSP. Another way to think about this is that $h(n)$ has infinite support for any of the ideal filters. Thus, it cannot be zero for $n < 0$ making it necessarily non-causal.
2. When implemented in software or hardware, due to the finite number of elements employed for processing, an ideal filter exhibits the undesirable Gibbs phenomenon. In signal and image processing, this shows up as the infamous ringing effect.

To bypass the above inadequacies, the design of digital filters using the windowing technique is an alternative to ideal filters. These filters can be implemented as FIR filters, and make use of the well-known Bartlett, Blackman, Hamming, Hanning, and Kaiser windows. The shortcoming of digital filter design via windowing techniques is that one cannot individually control the design parameters of the filter. For example, in Figure 1, ω_p (passband frequency) and ω_s (stopband frequency) cannot be independently controlled using the windowing technique. By making the transition band (frequencies between ω_p and ω_s) narrower, you must make a sacrifice in the form of larger undesirable ripples in the passband (frequencies between 0 and ω_p) and stopband (frequencies exceeding ω_s); that is smaller $\omega_s - \omega_p$ necessitates δ_1 and δ_2 larger. These tradeoffs are all due to bypassing the two inadequacies of ideal filters.

Filter Design and Analysis Tool

In this section, you will learn how to use MATLAB's handy *filter design and analysis tool* (**FDAT**) [3]. To begin, start MATLAB. Now, enter **`fdatool`** into the command window. This should, when executed, bring up the FDAT's GUI, shown in Figure 3 below. The process of designing a filter is fairly self-explanatory: you simply set all of the filter specifications in the lower half of the GUI. When you are satisfied with your specifications, click on the Design Filter button. The magnitude response of the resulting filter will appear in the Magnitude Response pane. Note that you can view the coefficients of the filter's transfer function in second order

sections by clicking on the Filter coefficients button at the top of the GUI (which looks like **[b,a]**).

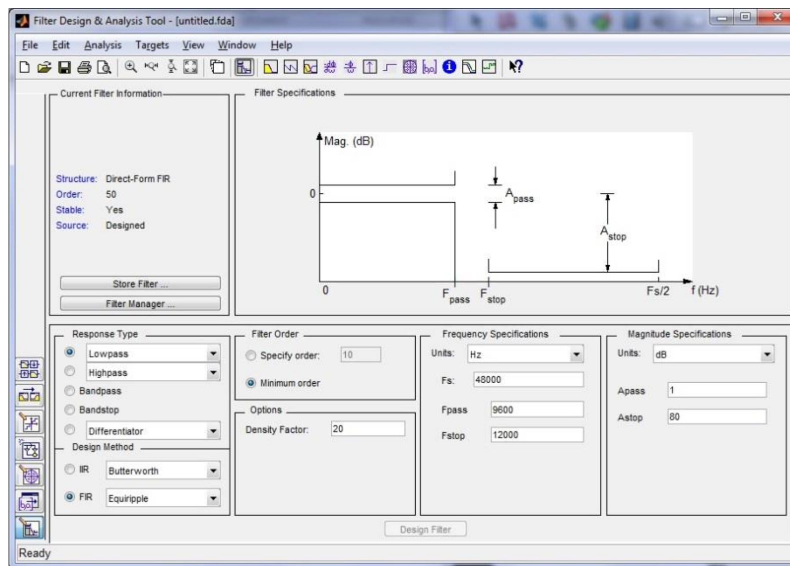


Figure 3. Filter Design and Analysis Tool

One useful feature of FDAT is that you can store multiple filters at once and switch between them as you wish. After you have designed a filter, you can store it by selecting the **Store Filter** button; this will prompt you to enter a name for the filter. Once you have stored the filter, you can begin designing a new filter by choosing new filter specifications. To access previously stored filters, click on the **Filter Manager** button. Finally, you can save all of your stored filters in one "**session**" by selecting **File→Save Session As**; this will save your session with a **.fda** extension. You can always open previously saved sessions in the FDAT GUI. Another feature of FDAT that we will be using is to export filters to a **Simulink** model as a single-input, single-output block. To do this, make sure the filter you want is currently shown in the GUI (if not, switch to it using the Filter Manager). Then, click on **File→Export to Simulink Model**; a new set of options should appear in the lower half of the GUI. Give the filter a good, descriptive **Block Name** and make sure the Destination is Current. Press the **Realize Model** Button (this only works if you have a Simulink Model currently open, of course).

Simulink

Simulink is an environment for simulation and model-based design for dynamic and embedded systems. It provides an interactive graphical environment and a customizable set of block libraries that let you *design, simulate, implement, and test* a variety of time-varying systems, including communications, controls, signal processing, video processing, and image processing [5]. In addition, you do not specify exactly how signals are computed in Simulink. You simply connect together blocks that represent systems. These blocks declare a relationship between the input signal and the output signal. The Simulink excels at modeling continuous-time systems. Of course, continuous-time systems are not directly realizable on a computer, so Simulink must ***simulate*** the system. The Simulink can also model discrete-time systems, and mixed discrete and continuous-time systems.

To open the Simulink library browser, first you need the MATLAB to be running. Start MATLAB, and then in the MATLAB Command Window, enter **`simulink`**. Then, the Simulink library browser opens (Figure 4). You can also open the Simulink Library Browser by clicking the Simulink icon on the MATLAB toolbar.

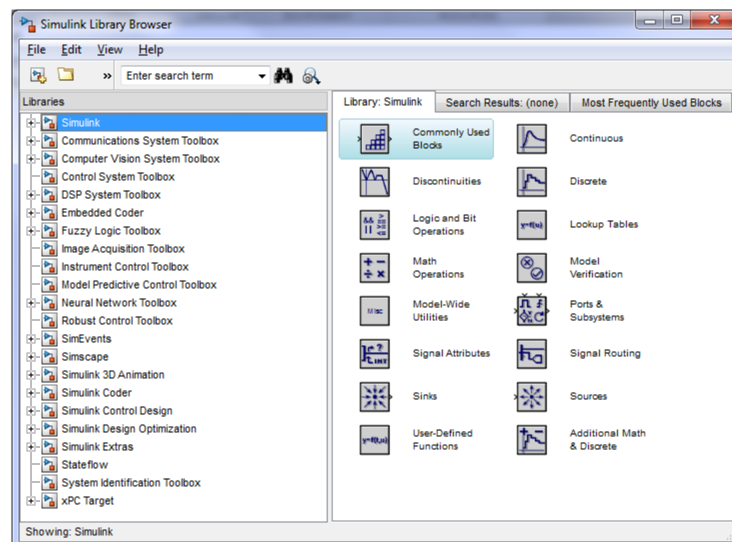


Figure 4. Simulink library browser

The Simulink library browser displays the block libraries installed on your computer. You can start building models by copying blocks from a library into a Simulink editor window. To create a new Simulink model;

1) From the Simulink library browser menu, select **File→New → Model**. An empty model opens in the Simulink editor window.
2) Select **File→Save**, in the Save As dialog box, enter a name for your model, and then click Save. Simulink saves your model with the file extension .mdl or .slx.

The Simulink editor window contains a block diagram of your model. You can build models in the editor by dragging blocks from the Simulink Library Browser window, arranging the blocks logically, connecting the blocks with signal lines, and then setting the parameters for each block. The Simulink editor window also allows you to:

- Set configuration parameters for the model, including the start and stop time, type of solver to use, and data import/export settings.
- Start and stop a simulation of the model.
- Save the model.
- Print the block diagram.

Before you simulate a model, you have to set simulation options. Specify options using the *Configuration Parameters* dialog box. In the Simulink editor window, select **Simulation → Configuration Parameters**. The Configuration Parameters dialog box opens to the Solver pane. After entering your configuration parameters, you are ready to simulate the simple model and visualize the simulation results. In the Simulink editor window, select **Simulation → Start** from the menu. The simulation runs, and then stops when it reaches the stop time specified in the Configuration Parameters dialog box. Alternatively, you can control a simulation by clicking the Start simulation button and Stop simulation button on the editor window toolbar.

Importing Audio into MATLAB Workspace

Representing, Playing and Plotting Sampled Sound Signals in MATLAB

In MATLAB, **mono** sounds can be represented as a really long vector, and **stereo** sounds as two really long vectors put together. You can edit and modify the vector in order to change the sound. It should be noted that the short sound signal in MATLAB requires a very long vector. For example, suppose we wanted to represent a **3-second** recording of a sound by *sampling* the sound (recording a value) once every **2 ms**. This means we are recording ***500** numbers every second*, so if we did it for 3 seconds we would need a vector of length **1500** to represent all the numbers in MATLAB. Fortunately, MATLAB can handle fairly big vectors. MATLAB has some sound signals already built-in. You will need your headphones for this part. To load and play one of these sounds, type

```
>> load handel
>> sound(y,Fs)
```

The command loads a variable y, which contains a vector of 73113 elements. This song is about 9 seconds long, with **8192** samples each second (the sampling rate is stored in the variable F_s). Recall that you can specify a part of a vector using a range of indices with the colon operator. Play the first 1/3 of the recording by typing:

```
>> L=length(y)
>> sound(y(1:L/3),Fs)
```

Sounds can be on your computer in different formats. For example, **.wav** and **.mp3** files are two particular formats for storing sounds, and sound-playing programs know how to read the files and produce sound using the computer's sound device. These formats all store a sampled signal, so the song is really just a long list of numbers, i.e. values of the signal at each sample time. There are a number of ways that we can get sounds into MATLAB, including:

- Convert an external sound file into a MATLAB vector, e.g. using the **`wavread`** command for sound files stored in the **.wav** format, as in either:

```
>> [mySound Fs] = wavread('somesound.wav');
>> [mySound Fs] = wavread('somesound');
```

- Load a sound signal that already exists as a MATLAB vector into your workspace, using the **load** command. The syntax is either:

```
>>load handel;
>> load('handel');
```

(The signal and sampling frequency are put into previously-defined variables, in this case y and F_s. The usual value of F_s for built-in MATLAB sounds is 8192 Hz.)

- Create a vector from scratch in MATLAB.
- Use the **wavrecord** function in MATLAB to record sound for the audio input of your sound card.

We will use one of these methods to generate vectors which store sounds. Once you have generated a vector, the original format doesn't matter, so plotting and playing sounds is the same for all cases. Format will matter again when you want to "write out" the sound (save it in a file for future use). You can play a vector as a sound using the **sound** command. This command requires values to be in the range $[-1, 1]$ or it will clip them to this range. (You can use **soundsc** instead to automatically scale values to this range. This avoids clipping, but changes the loudness of the sound.) If you want to play a sampled sound (in MATLAB or with other tools), you need to specify the playback rate (sampling rate) F_s in samples per second (Hz). (Recall $F_s = 1/T_s$, where T_s is the time between samples). In MATLAB, the function **sound** allows you to specify the sampling frequency as the second argument:

```
>> sound(mySound,Fs);
```

If you don't specify anything, it will use the default sampling frequency of 8192Hz (which was just fine for the Handel example). To convince yourself that this is important, see the prelab question1. You can plot the time signals using either **plot** or **stem**, where **plot** hides the discrete nature and **stem** makes it explicit. When you are plotting sampled signals as time functions, you should make sure that the appropriate time information is displayed on the time axis, for example using:

```
>> t=0:Ts:2;
>> plot(t,y);
```

where $T_s = 1/F_s$. Another thing to keep in mind is that a very long signal will be hard to view in a single plot, especially a sinusoid with constant amplitude. It is often useful to plot only portions

of a signal, or plot the signal in sections. You can display multiple plots at once using the function **subplot(n,m,k)**, which creates a $n \times m$ matrix of plot spaces in the figure, you can save it in the **.wav** format by using:

```
>> wavwrite(mySound,Fs,'filename');
```

where **mySound** is the vector your sound is stored in, F_S is the appropriate sample rate, and "**filename.wav**" is the name you want the file to have. If you have values outside $[-1,1]$, they will be clipped and the function will return a warning message. If you don't specify the sampling frequency, it will assume a default of 8192Hz. You can also specify the number of bits/sample, but we will just stick with the default (16 bits). If you want to write a stereo file, the format should be a matrix with 2 columns, one per channel.

Example 2. Reading stereo sound:

```
[audio,fs]=wavread('name_of_wav_file.wav'); % transfers .wav
file into array audio and reading the sampling frequency to fs
x = audio ( : , 1 ); % assign left channel data to x
y = audio ( : , 2 ); % assign right channel data to y
Ts=(1/fs)*length(x); % Calculates the duration of .wav file
t=linspace(0,Ts,length(x)); % Interpolates 't' from 0 to 'time'
so creates a time vector
sound(audio,fs) % Plays the array audio with a sampling rate fs
```

Since sound signals are represented as vectors in MATLAB, you can do any mathematical operation on the sound signals that you could do on elements in a vector. In other words, you can create your own sounds with MATLAB scripts and functions. You will get to make a sound composition by modifying, mixing and stringing sounds together. (**doc soundmixer)**.

Importing a Multimedia File to Simulink

In Simulink library, there is a specific block to read multimedia files named as *From Multimedia File*. Any sound or image data supported by MATLAB can be imported into Simulink model by using this block. Loaded audio file can be listened with the block named *To Audio Device*. If the loaded multimedia file is image or video, the *Video Viewer* block should be used.

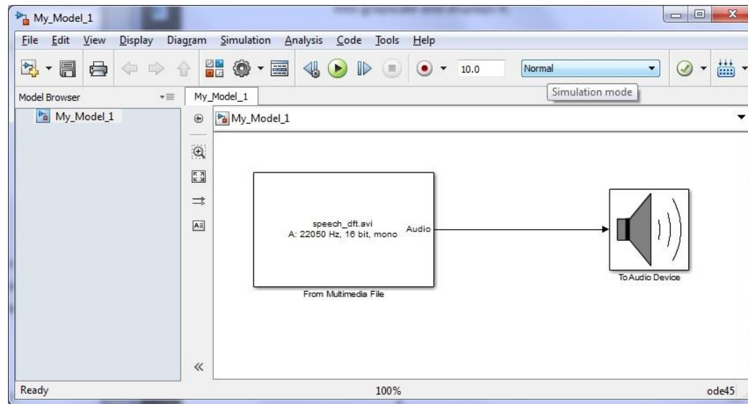


Figure 5. Import audio file to Simulink model

Prelab

Question 1. The MATLAB function sound (see help sound) with syntax

```
>> sound(sampledSignal, frequency)
```

sends the one-dimensional array or vector **`sampledSignal`** to the audio card in your computer. The second argument specifies the sampling frequency in Hertz. The values in **`sampledSignal`** are assumed to be real numbers in the range $[-1, 1]$. Values outside this range are clipped to -1 or 1.

Download the audio file **"lab_5_Audio_1.wav"** from the course directory **/groups/e/elec364_1** (see **Laboratory Guidelines**).

Use wavread command to read the audio file **"lab_5_Audio_1.wav"**

```
>> [y,fs]=wavread('lab_5_Audio_1.wav');
```

After reading, listen to

a)

```
>> sound(y,fs)
```

b)

```
>> sound(0.25*y,fs)
```

and

```
>> sound(4*y,fs)
```

Explain in what way these are different from what you heard in the part (a).

c)

```
>> sound(y, fs/2)
```

and

```
>> sound(y, fs*2)
```

Explain how these are different from what you heard in the part (a).

Question 2. Designing the Filters Using FDATOOL, perform the following tasks, assuming a sampling rate of 8 kHz:

I. Design a minimum order, stable, lowpass Butterworth filter with a passband frequency of 1 kHz and a stopband frequency of 1.4 kHz. Make the attenuation 1 dB at the passband frequency and 80 dB at the stopband frequency.

II. Design a minimum order, stable, lowpass Chebyshev Type I filter with the same specifications as the Butterworth filter.

III. Design a lowpass FIR filter using the Blackman Window with a cutoff frequency of 1 kHz. Specify the order of the filter such that the first minimum in the stopband (preceding the first lobe) is as close to 1.4 kHz as possible without exceeding it.

Now answer the following

a) What is the order of the lowpass Butterworth filter you designed?
b) What is the order of the lowpass Chebyshev Type I filter you designed?
c) Compare the implementation cast of each of the 3 filters. Do you see how inefficient the windowing technique is? How much more expensive in terms of memory is the windowing technique from the best IIR filter?
d) For the filter you designed in Part I, round the filter coefficients (b,a) to the nearest integer value. Use the MATLAB round command:

```
>> help round
```

```
round   rounds towards the nearest decimal or integer

round(X) rounds each element of X to the nearest integer.

round(X, N), for positive integers N, rounds to N digits to
the right of the decimal point. If N is zero, X is rounded to
the nearest integer.  If N is less than zero, X is rounded to
the left of the decimal point. N must be a scalar integer.
```

Obtain the pole-zero plot of the filter with the original coefficients (use the zplane(b,a) command where b and a are the filter coefficients) and the pole-zero plot of the filter with the rounded values of the coefficients. Comment on any differences between the two plots.

In this exercise, we intentionally rounded the coefficients using the MATLAB round command. As alluded to in Example 4 of Lab 2, (unintentional) rounding may result from the way real numbers are represented inside a computer. Here is Example 4 from Lab 2:

```
clear

x = 0.1 + 0.1 + 0.1

if ( x == 0.3)

 disp(‘ x is equal to 0.3’)

else

 disp(‘x is is NOT equal to 0.3’)

end

% use the fprintf output command to display the value of x to
% 18 decimal place  in a field-width of 21 columns

fprintf(‘ x is really = %21.18f\n’, x)
```

Run the above MATLAB code and see the effects of rounding for yourself. Comment on the source of the rounding error when a real number is stored inside a computer. HINT: IEEE 754 floating point representation. ANOTHER HINT : 1/3 = 0.33333333333333333333…

Question 3. Creating a Noisy Signal using Simulink

You will create a sinusoid single corrupted by high frequency noise. Both the sine wave and the noise will be discrete; thus, your whole Simulink model will be discrete (this is purposely done to simplify things). Here's how to create such a signal:

I. Add a Sine Wave block to your model. Set the frequency to 100 Hz and the Sample time to 1/8000.

II. Add a Uniform Random Number block to your model. Set Sample time to 1/8000 and Maximum and Minimum to 5 and -5, respectively. As its name suggests, this block outputs random numbers, which for our purposes is a good model for noise.

III. We want to restrict our noise to only have high frequency components. To do this, we are going to send it through a highpass filter. Add a Highpass Filter block and open the Block Parameters. Under Filter specifications, set Impulse response to IIR and Order mode to Minimum. Under Frequency specifications, set Frequency units to Hz, Input Fs to 8000, Fstop to 3600, and Fpass to 3800. Under Magnitude specifications, set Magnitude units to dB, Astop to 60, and Apass to 1. Finally, under Algorithm, set Design method to Elliptic and Structure to Direct-form II SOS. Right now, all that you have to understand about this filter is that it only passes frequencies above 3800 Hz (Fpass); anything below 3600 Hz (Fstop) is attenuated by at least 60 dB (the region between these two frequencies is a transition region).

IV. Send the random number signal from Step II through the highpass filter and add it to the sine wave you made in Step I.

V. To view the noisy sine wave signal, use Scope blocks. Also, use another scope to view the original sine wave. Use the Scope parameters to set 'limit data points to last' parameter and to save the data to your workspace.

Useful MATLAB Commands

You can use MATLAB **`sim`** *command to Run Simulink model from the script file.*

Tools: **`Sumlink, sptool, fdatool`**

Sound: **`wavread, wavwrite, sound, soundsc, soundmixer, audioplayer`**
Filter: **`butter, buttap,cheby1, cheb1ap, cheby2, cheb2ap, ellip, ellipap, besselap`**

Frequency Domain: **`fft`**

Signal manipulation: **`filter, filtfilt`**

Elementary Matrices and Matrix Manipulation: **`i, ones, pi, rand, randn, zeros.`**

Two-Dimensional Graphics: **`axis, grid, legend, plot, stem, title, xlabel, ylabel.`**

Elementary Functions: **`cos, sin, exp, imag, real,abs.`**

Data Analysis: **`sum`**

Lab problems

During the lab, you must show your results/outputs of each problem to your Lab demonstrator/TA. After the lab, each student must submit a lab report contains results, discussion, MATLAB codes and Simulink models. Also, at the end of the lab report attach the results and solutions to the prelab questions.

Problem 1. Consider the input signal

$$x(t) = \sin(100t) + \sin(2000t) + \sin(6000t)$$

We need to pass the term $\sin(2000t)$ and to attenuate the other terms.

a) Design a digital fourth and eighth-order IIR Butterworth filters. Plot the input signal along with the magnitude response of the filter and its output signal for each filter.
b) Design a digital fourth and eighth-order IIR Chebyshev type I filters. Plot the input signal along with the magnitude response of the filter and its output signal for each filter. (consider $R_p = 1\ dB$).
c) Design a digital fourth and eighth-order IIR Elliptic filters. Plot the input signal along with the magnitude response of the filter and its output signal for each filter. (consider $R_p = 1\ dB, R_s = 60\ dB$)
d) For parts a), b), and c) plot the discrete Fourier transform of the input signal and the output signal for eighth order filters, you can use **`FFT`** command.

Hint:

- You have to choose a sampling frequency.
- When you use the MATLAB functions **`butter, ellip, cheby1`**, we need to normalize this digital passband to lie in the interval [0 1].

Problem 2.

Part A: Create a signal with three seconds duration. The signal contains three tones of equal amplitude, as following

a) Contain a 200 Hz tone for $0 < t < 3$.
b) Contain a 330 Hz tone for $1 < t < 3$.
c) Contain a 480 Hz tone for $2 < t < 3$.

Note: The signal should have a sampling frequency equal to 8192 Hz.

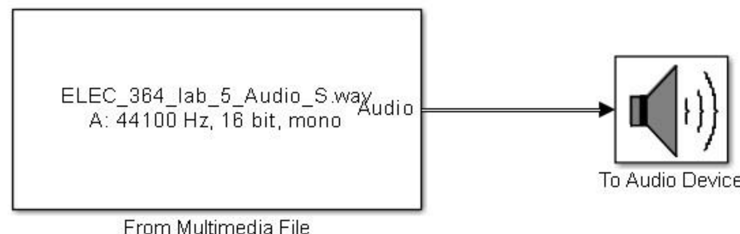
View the signal before proceeding (in time domain and in frequency domain), and listen to the signal using `soundsc` command.

Part B: Design a filter to remove the 330 Hz component from the main signal.

a) Plot the designed response (magnitude and phase) of the filter.
b) Filter the main signal with the designed filter.
c) Compare signals and spectra (in time domain and in frequency domain) before and after filtering, and listen to the filtered signal using soundsc command.

Problem 3. Building Simulink model to read an audio file and manipulate the sound by adding upsamping or downsampling block.

a) Download the audio file "**ELEC_364_lab_5_Audio_S.wav**" from course directory **/groups/e/elec364_1** (see **Laboratory Guidelines**).
b) Open new Simulink model window, and import the audio file "**ELEC_364_lab_5_Audio_S.wav**". Set **Audio output data type** to double.
c) Add **To Audio Device** block and connect the output of the audio block to the input of the **To Audio Device** block. Click run button and listen.



Part A:

d) After listening, add **Downsample** block, and set **downsample factor k** to 8.

e) Build the Simulink model shown in Figure 6. Set the Scope parameter “limit data point to last” to 327680.

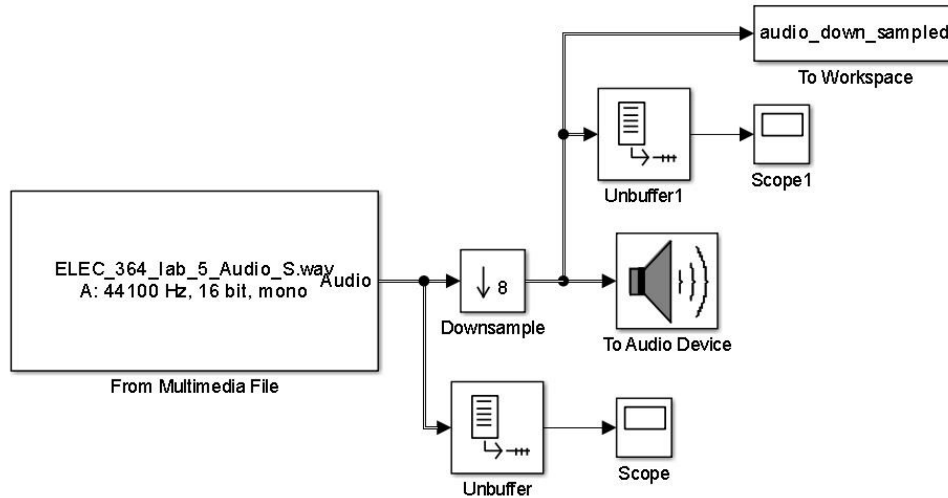


Figure 6. Problem 3-Part A, Simulink Model

f) Click run button, and listen to the sound. Did you notice any difference; write that in your lab report. Use single figure to plot the original sound signal in time and frequency domain (use FFT) and **audio_down_sampled** signal in time and frequency domain (use FFT), you output results should be similar to Figure 7.

g) Now, reset the **downsample factor k** to 16 and repeat the step f).

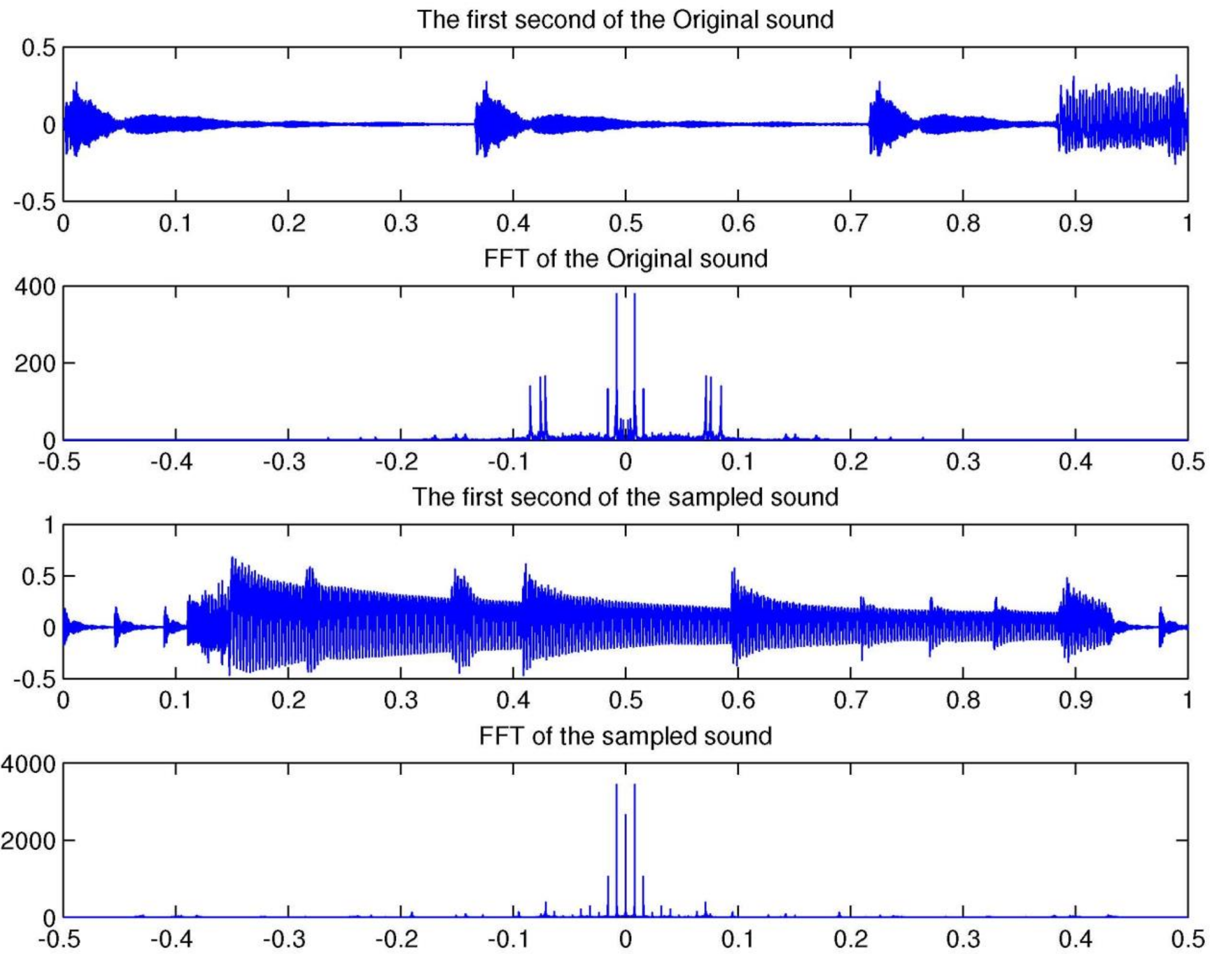


Figure 7. Problem 3-part A, plot of the original audio signal and the down sampled audio signal in time and frequency domain.

Part B:

h) Build the Simulink model shown in Figure 8, and add Upsample block. Set the **upsample factor L** to 8, and set the Scope parameter “limit data point to last” to 327680.

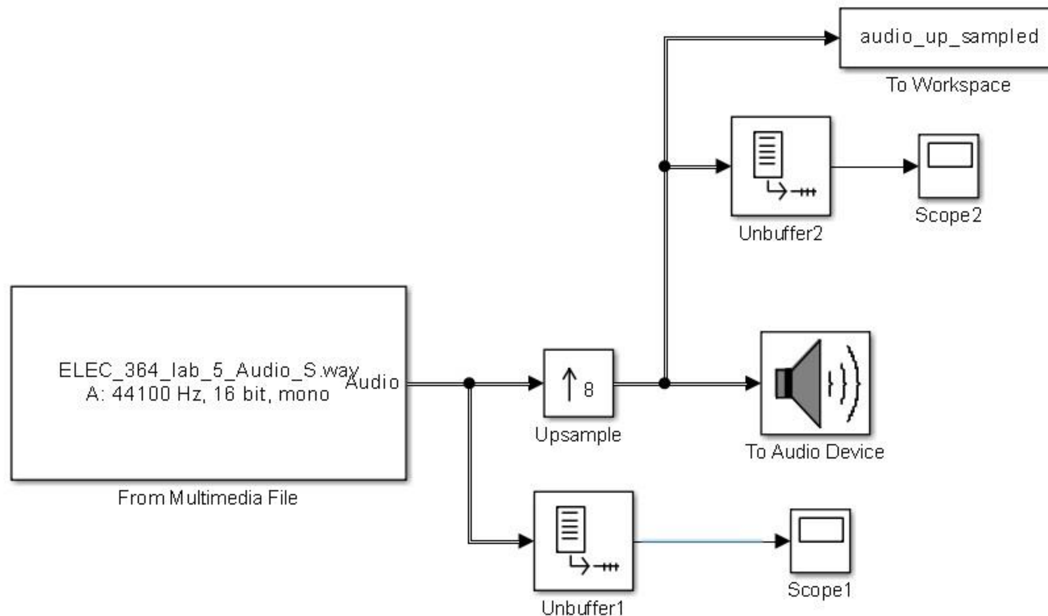


Figure 8. Problem 3-Part B, Simulink model

i) Double click of the **To Audio Device** block, and uncheck the **Inherit sample rate from input** option.

j) Click run button, and listen to the sound. Did you notice any difference; write that in your lab report. Use single figure to plot the original sound signal in time and frequency domain (use FFT) and **audio_up_sampled** signal in time and frequency domain (use FFT), you output results should be similar to Figure 9.

k) Set the **upsample factor L** to 16 and repeat the step j).

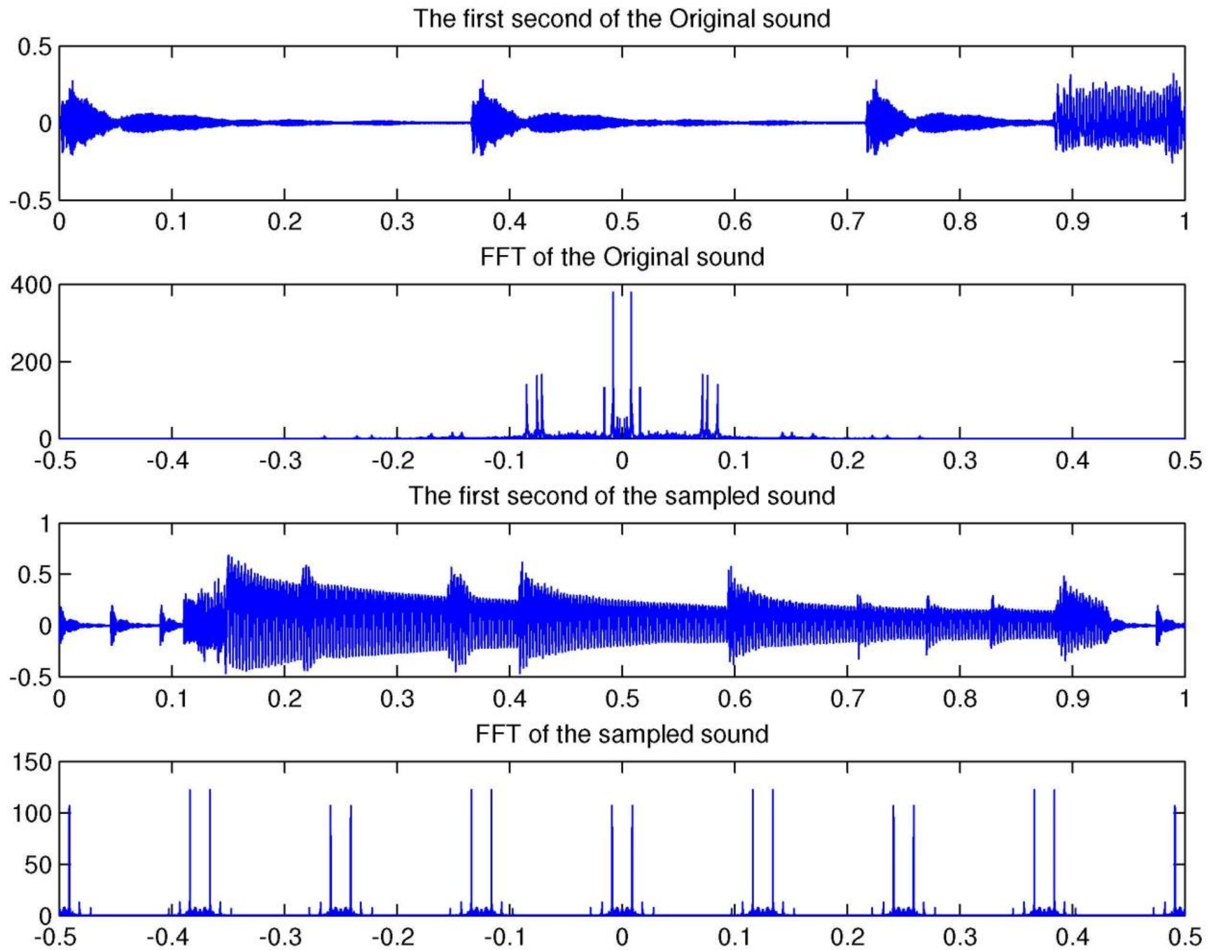


Figure 9. Problem 3-part A, plot of the original audio signal and the up sampled audio signal in time and frequency domain.

References

[1] Signals and Systems, 2nd ed. , A.V. Oppenheim and A.S Willsky, Prentice-Hall, ISBN 0-13-814757-4, 1997.

[2] http://www.mathworks.com/help/signal/ug/sptool-an-interactive-signal-processing-environment.html.

[3] http://www.mathworks.com/products/signal/code-examples.html?file=/products/demos/shipping/signal/introfdatooldemo.html.

[4] http://www.mathworks.com/discovery/filter-design.html.

[5] http://www.mathworks.com/academia/student_center/tutorials/simulink-launchpad.html.