Continuous Time Linear Systems Laboratory (EE 235)

Collection Editor:

University Of Washington Dept. of Electrical Engineering

Continuous Time Linear Systems Laboratory (EE 235)

Collection Editor:

University Of Washington Dept. of Electrical Engineering

Authors:

University Of Washington Dept. of Electrical Engineering
UW EE235 TA UW EE235 TA

C O N N E X I O N S

Rice University, Houston, Texas

Table of Contents

iv

Chapter 1

Lab 1

1.1 Introduction to MATLAB and Scripts[1]

1.1.1 Introduction

The goal of this lab is to provide exercises that will help you get started learning MATLAB. You will learn about the help function, vectors, complex numbers, simple math operations, and 2-D plots. You may find it useful to try some of the built-in demos in Matlab. Type `demo` to see the choices. In particular, look at the demo on "Basic matrix operations" (under "Mathematics") and on "2-D" plots (under "Graphics"). We will also look at script files in MATLAB, which we will refer to as M-files and have the file extension `*.m`.

1.1.2 Getting Started

Start MATLAB by clicking on it in the start menu. Once MATLAB is running you will see a screen similar to Figure 1. The command window, (A), is where you can enter commands. The current working directory, (B), displays the directory that MATLAB will look first for any commands. For example, if you made a new MATLAB function called `myfunc.m`, then this will need to be placed in the current working directory. You can change the working directory by typing it in the box, clicking the "..." button to browse to a folder, or typing `cd [directory name]` (i.e. `cd 'H:\ee235\lab0\'`), which is similar to the DOS/Linux cd command).

> NOTE: MATLAB supports tab completion. This means that you can type part of a command and then press tab and it will fill in the rest of the command automatically.

The workspace displays information about all the variables currently active and is shown in (C). The files in the current directory can also be displayed in (C) by clicking on the tab labeled `Current Directory`. A history of your commands is shown in (D). If you find that you do not need some of these windows open you can close them by clicking on the small x in that section of the window.

---

[1]This content is available online at <http://cnx.org/content/m13554/1.18/>.
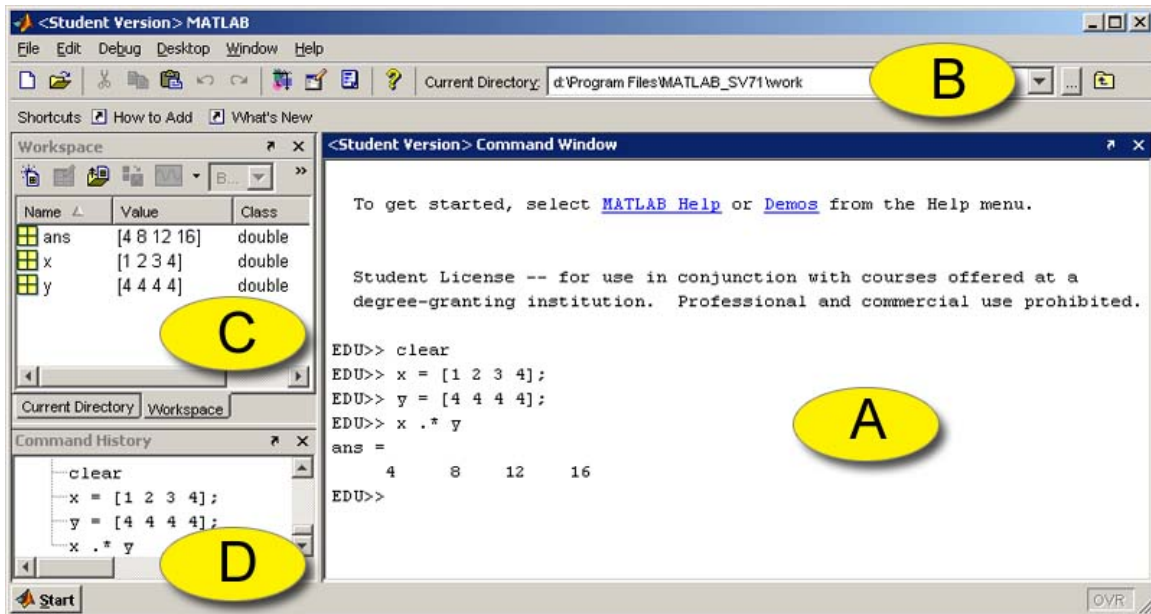
The MATLAB GUI

Figure 1.1: (a) Command window, (b) working directory, (c) workspace, (d) command history.

NOTE: There are a number of different ways to use MATLAB on Linux. Typing `matlab` at the command prompt will run MATLAB in X-Windows (warning, MATLAB in X-Windows can be slow when connecting off campus). To run MATLAB without X-Windows type `matlab -nodisplay`. You can also run MATLAB using the current terminal for commands and use X-Windows for everything else (like figures) by typing `matlab -nodesktop`.

1.1.3 MATLAB Commands

MATLAB works with matrices and therefore treats all variables as a matrix. To enter the matrix

$$x = \left(\begin{array}{ccc} 3 & 1 & 5 \\ 6 & 4 & 1 \end{array} \right)$$

type the command `x = [3 1 5; 6 4 1]`. We can represent an array with a vector, which is a special case of a matrix. A vector can be entered in by typing `y = [1 2 3]`. Now try entering `z = [1 2 3]'`. Is the output what you expect?

1. Familiarize yourself with the help command. Typing `help` gives you a list of all help topics. Typing `help <topicname>` gives help on a specific MATLAB function. For example, use `help plot` to learn about the plot command.

    Rule 1.1: More useful commands

    - `whos` lists all variables
    - `clear` clears all variables

2. Perform the following operations in MATLAB:

a. Generate the following column vectors as MATLAB variables: $x = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$ and $y = \begin{pmatrix} 3 \\ 1 \end{pmatrix}$

b. Using the computer, issue the following MATLAB commands

```
x * y'
x' * y
x .* y
```

Be sure you understand the differences between each of these and you know what the ', *, and .* operators do.

c. Convince yourself that the answer makes sense by checking the matrix dimension and computing each result by hand.

Rule 1.2: Plot and Subplot

The Matlab command `plot` allows you to graphically display vector data (in our case here, the two signals). For example, if you had the variables `t` for time, and `y` for the signal, typing the command `plot(t, y);` will display a plot of `t` vs. `y`. See `help plot` for more information.

Annotating your plots is very IMPORTANT! Here are a few annotation commands.

3.
- `title('Here is a title');` - Adds the text "Here is a title" to the top of the plot.
- `xlabel('Control Voltage (mV)');` - Adds text to the X-axis.
- `ylabel('Current (mA)');` - Adds text to the Y-axis.
- `grid on;` - Adds a grid to the plot.

In order to display multiple plots in one window you must use the `subplot` command. This command takes three arguments, `subplot(m, n, p)`. The first two breaks the window into a m by n matrix of smaller plot windows. The third argument, p, selects which plot is active. For example, if we have three signals x, y, and z, that we want to plot against time, t, then we can use the subplot command to produce a single window with three plots stacked on top of each other.

```
subplot(3,1,1);
plot(t,x);
subplot(3,1,2);
plot(t,y);
subplot(3,1,3);
plot(t,z);
```

See `help subplot` and `help plot` for much more information.

Create and plot a signal $x_0(t) = te^{-|t|}$ over the time range [-10,10] using the following MATLAB commands:

```
t = -10:0.1:10;
xo = t .* exp(-abs(t));
plot(t, xo); grid;
```

The first command defines an array with time values having an 0.1 increment. The ";" is used to suppress printout of the arrays (which are large), and the **"grid"** command makes the plot easier to read. Now create the signals:

$$x_e(t) = |t|e^{-|t|} \tag{1.1}$$

$$x(t) = 0.5 * [x_0(t) + x_e(t)] \tag{1.2}$$

Plot all signals together using 3 plots stacked on top of each other with the subplot command.

```
subplot(3,1,1);
plot(t,xo);
subplot(3,1,2);
plot(t,xe);
subplot(3,1,3);
plot(t,x);
```

Note that $x_o(t)$ and $x_e(t)$ are the odd and even components, respectively, of $x(t) = te^{-t}u(t)$

4. **Complex Numbers:** One of the strengths of MATLAB is that most of its commands work with complex numbers. Perform the following computations in MATLAB.

   a. MATLAB recognizes i as an imaginary number. Try entering `sqrt(-1)` into MATLAB, does the result make sense?

   b. MATLAB uses the letter i instead of j by default. Electrical Engineers prefer using j however, and MATLAB will recognize that as well. Try entering `i+j`, does this make sense.

   > NOTE: If you are using complex numbers in your code, it's a good idea to avoid using i and j as variables to prevent confusion.

   c. Define $z_1 = 1 + j$. Find the magnitude, phase, real and imaginary parts of z (using `abs()`, `angle()`, `real()`, `imag()`, respectively). Is the phase in radians or degrees?

   d. Find the magnitude of $z_1 + z_2$ where $z_2 = 2e^{\frac{j\pi}{3}}$

   e. Compute the value of j^j . Is the result what you expect?

5. **Complex Functions:**   MATLAB also handles complex time functions in the same way (again, implemented as vectors) . Create a signal $x_1(t) = te^{jt}$ over the range [-10,10], as in part 3. Next plot the real and imaginary parts of the signal in two plots, one over the other using the subplot command. Notice that one plot is odd and one is even. Try proving to your self analytically that this is what you would expect.

6. **Playing and Plotting a Sound** Load the built-in data named "handel" and play it:

```
load handel;
plot(linspace(0,9,73113),y);
sound(y);
```

> NOTE: You can use the `clear` command in MATLAB to clear all of the varibles

1.1.4 Script Files

Scripts are `m-files` files that contain a sequence of commands that are executed exactly as if they were manually typed into the MATLAB console. Script files are useful for writing things in MATLAB that you want to save and run later. You can enter all the commands into a script file that you can run at a later time, and have the ability to go back and edit it later.

You need to use a text editor to create script files, e.g. `Notepad` on the PC's (`pico`, `emacs`, or `vi` on Linux machines). MATLAB also has an internal editor that you can start by clicking on a `.m` file within MATLAB's file browser. All are easy to learn and will produce text files that MATLAB can read.

Click here[2] to download the `dampedCosine.m` script and be sure to save it with that name to follow the instructions here exactly. It is very important that script filenames end in `.m`. Be sure that MATLAB's working directory is set to the location of where you saved the script file. Type dampedCosine at the MATLAB prompt. Look at the m-file in a text editor and verify that you get the plot predicted in the comment field of the script.

NOTE: The `%` character marks the rest of the line as a comment.

Exercise 1.1.1
Scripts
Now we are going to edit the `dampedCosine.m` file to create our own script file. At the top of the file, you will see the following commands

```
diary 'your_name_Lab1.txt'
disp('NAME: your name')
disp('SECTION:your section')
```

1. Edit the dampedCosine.m (download from link above) script and enter your name and section where indicated. Save this new version of the script as `yourName_dampedCosine.m`
2. Edit the script to create a second signal where the cosine with twice the period (which gives half the frequency) of the first.
3. Add to the script the commands to plot these together with the first signal on top and the second on the bottom. In other words, you should have a single figure with two different plots, one on top and one on bottom. You will need to use `subplot` and `plot`. Save this plot as `yourName_dampedCosine.fig`.
4. Show the TA your dampedCosine plot. What is the period of the cosine?

Exercise 1.1.2
Complex exponentials
Download and run compexp.m[3] , which includes a 3-D plot of a complex exponential, $y(t)$, as well as 2-D magnitude/phase and real/imaginary plots. You need 2 2-D plots to have the same information as the 3-D plot. How would you change the script to make the oscillation frequency lower by half? How would you change the script to make the decay faster? Show the TA your plots.

---

[2]http://cnx.org/content/m13554/latest/dampedCosine.m
[3]http://cnx.org/content/m13554/latest/compexp.m

Chapter 2

Lab 2

2.1 Functions in MATLAB and the Groove Station[1]

2.1.1 Introduction

In this lab, you will learn about MATLAB function files, which we will refer to as m-files and have the file extension `*.m`, the same as script files. You will create a number of functions for manipulating sound signals to create a groove (a short song).

The difference between a function and a script is that functions can return values and take parameters, while scripts are simply a collection of commands. Unlike script files, any variables created in the function are not available after the function has run, that is, the variables are active inside the scope of the function, and the variables cannot be accessed out-of-scope. The MATLAB on-line help system has a nice write-up about functions and how to handle various things like returning more than one value, checking the number of arguments, etc. To learn more, type the following commands and read the online help:

```
≫ help function
≫ help script
```

You need to use a text editor to create function files. MATLAB has an internal editor that you can start by clicking "File" and then "New" "m-file". You can also use other editors such as `Notepad` on the PC's, and `pico`, `emacs`, or `vi` on Unix machines.

> NOTE: Remember that in order to run custom scripts or functions, the MATLAB working directory needs to be set to the location of those files.

2.1.2 Sound in MATLAB

Download the sound samples from the sound resources (Section 2.2) page and save them to your working directory. Use `wavread` to load `.wav` files, and use `load` to load `.mat` files. Plot each one in turn, and try to guess what it will sound like (the name might help).

On a computer, sounds are represented digitally, which means that only samples of the signal at fixed time intervals are stored. We'll learn more about this later. For now you just need to know that the time interval Ts (or equivalently the sampling rate Fs=1/Ts) is something you need to keep track of for playing sounds.

---

[1]This content is available online at <http://cnx.org/content/m13555/1.25/>.

Now play each sound. The goal is to learn how the time domain signal sounds. Use the sound command to play a sound signal. You must specify the playback sample rate (Fs), which will be the same as the sample rate of the sound samples on the web site (they are 8000 Hz). For example, if you wanted to play a sound called bell and its sample rate was 8000 Hz, then you would enter the following command,

```
    ≫ Fs=8000;
≫ sound(bell, Fs);
```

If you use a different value for Fs, you will effectively be doing time scaling.

When working with sound in MATLAB, it is important to remember that the values of the audio signals are in the range [-1, 1]. Keep this in mind when you are writing your functions. Your functions should expect inputs with values in the range [-1, 1] and anything out of that range will be clipped when you play the sound.

2.1.3 Function Files

Before we can create our groove, we need to make functions that will allow us to modify the sound signals in various ways. After all, wouldn't it be boring to make a groove out of the same note over and over again? Let's create some functions that will let us time scale, reverse, delay, fade, and repeat a sound, and mix two sounds together.

There are many functions built into MATLAB. One that will be useful here is `fliplr`, which is a one step way of time reversing a signal. Try this with the bell sound.

Another function that we created for you is timescale.m[2] , which you can use to speed up or slow down a signal. Download it and give it a try. Notice that it also changes the pitch of a sound − why?

Download the function fade.m[3] , make sure you save it as `fade.m`. Start MATLAB, and go to the directory where you saved the function. You can see and change your current directory at the top of the MATLAB screen. Enter "help fade" at the MATLAB prompt. If you did everything correctly, you should see the help text (in the `.m` file) in response to `help fade`. Notice that we've now added a new command to MATLAB that can be used as if it were a built-in function.

Enter the following commands at the MATLAB prompt:

```
    ≫ time = 0:.01:1;
≫ y = cos(time .* pi .* 25);
≫ plot(time, fade(y));
```

You can see in the plot that fade does indeed fade-out the cosine wave. You can use this function on audio signals as well.

Exercise 2.1.1 Fader

1. Modify the fade function so that you can adjust the slope of the ramp which will affect the level of the fade. Use the variable `level` (which is already in the parameter list for you in the function) to represent the strength of the fade as a decimal fraction. The function should make sure that the value is between 0 and 1.
2. Like in the code example above, plot your function with the cosine wave to see its effect. Throughout this lab you may find it helpful to plot functions (use the plot command).
3. Demonstrate to the TA that your fade function works.

Exercise 2.1.2 Repeater

---

[2]http://cnx.org/content/m13555/latest/timescale.m
[3]http://cnx.org/content/m13555/latest/fade.m

1. Create a function that repeats a sound N times. Use a `for` loop for this. Inside the `for` loop you will need to concatenate sound signals. For example, if you have two vectors `x` and `y`, you can concatenate them like this:

```
≫ x = [1 4 2 2 3];
≫ y = [5 8 3 9 0];
≫ x = [x y];
```

The first line of your function might look like this:

```
function [ out ] = repeat(in, N)
```

2. Demonstrate your repeater using an N specified by the TA.
3. Optional: Add an argument that let's you insert silence in between each repetition.

Exercise 2.1.3 Delay (Shift)

1. Create a function to time-delay a signal. Because we are working with digital data, you can do this by simply adding zeros (zero pad) in the front. The inputs to the function should be the signal and the amount of time-delay. The number of zeros to add will depend on the time-delay and the sample rate. The sound signals from the resource page have a sample rate of 8,000 Hz, but it is good coding style not to assume this and to still have the sample rate (Fs) be an input to the function in case you wanted to change it later.
2. Demonstrate to the TA that your delay function works by plotting the original and delayed signal together with the subplot command.

Exercise 2.1.4 Mixer

1. Create a function that adds two sound vectors together; your function should be able to handle inputs that are not the same size. The output values cannot be outside of the range [-1, 1], so you will have to re-scale them. One option is to re-scale the summed sound if it goes out of this range. You may want to look at the source code to the `soundsc` function for a way to do this. What happens if you let the sounds go out of this range and you try to play them with the `sound` command?

   NOTE: You can view the source code to most MATLAB functions by using the command `type function_name`.

2. Demonstrate to the TA that your mixer function works by playing a mix of two sounds.

2.1.4 Groove Station

In order to create a groove, you're going to need some instruments. The groove will be made up of some sound samples modified in any way you want and concatenated together to make one long sound vector. Use only the sound samples from the sound resources (Section 2.2) page.

Exercise 2.1.5 Make Your Groove

1. Create a script (not a function) to build your groove. You can use any combination of the above functions, or even create additional functions if you want. Use concatenation to combine the sounds together to make your groove. When you are finished save your groove with the `wavwrite` command (Remember to specify the sample rate (Fs), which for the sounds on the resource page is 8000 Hz).

2. Your groove should be between 10 and 30 seconds long.
3. Plot your newly created groove signal.
4. Demonstrate your groove to the TA. Explain how you created it.

2.2 Sound resources[4]

2.2.1 Sound Resources

Sound files in MATLAB and WAV format at 8000 Hz.

- blueslick.mat[5]
- doit.mat[6]
- fall.mat[7]
- shake.mat[8]
- tag.mat[9]
- rainstick.mat[10]
- bassdrum.wav[11]
- bleeep.wav[12]
- hatclosed.wav[13]
- snare.wav[14]

The University of Washington department of Electrical Engineering thanks Matt Swihart for the trumpet recordings and Tigh Bradley for the drum samples.
    Sound files in MATLAB and WAV format at 44100 Hz.

- castanets44m.wav[15] (modified from Prof. Sheila Hemami, Cornell University)

2.2.2 Additional Sound Resources

2.2.2.1 Sound files for Fourier Series and Gibbs Phenomenon lab

- trumpet.mat[16] 11025 Hz (UW EE thanks trumpeter Ed Castro for this clip)

2.2.2.2 Sound files for Filtering Periodic Signals lab

- mixed.wav[17] 8000 Hz

---

[4]This content is available online at <http://cnx.org/content/m13854/1.7/>.
[5]http://cnx.org/content/m13854/latest/blueslick.mat
[6]http://cnx.org/content/m13854/latest/doit.mat
[7]http://cnx.org/content/m13854/latest/fall.mat
[8]http://cnx.org/content/m13854/latest/shake.mat
[9]http://cnx.org/content/m13854/latest/tag.mat
[10]http://cnx.org/content/m13854/latest/rainstick.mat
[11]http://cnx.org/content/m13854/latest/bassdrum.wav
[12]http://cnx.org/content/m13854/latest/bleeep.wav
[13]http://cnx.org/content/m13854/latest/hatclosed.wav
[14]http://cnx.org/content/m13854/latest/snare.wav
[15]http://cnx.org/content/m13854/latest/castanets44m.wav
[16]http://cnx.org/content/m13854/latest/trumpet.mat
[17]http://cnx.org/content/m13854/latest/mixed.wav

Chapter 3

Lab 3

3.1 Convolution[1]

3.1.1 Introduction

In this lab, we will explore convolution and how it can be used with signals such as audio.

Since we are working on a computer, we are working with finite-length, discrete-time versions of signals. It is important to note that convolution in continuous-time systems cannot be exactly replicated in a discrete-time system, but using MATLAB's `conv` function for convolution, we can explore the basic effects and gain insight into what is going on. (You can learn more about discrete-time convolution in the UW EE 341 class.)

When you are explicitly working with discrete-time signals, you would plot them with `stem`. However, since we want to think of these as continuous time, we'll still use the `plot` command. An artifact that you may notice is that discontinuities (as in a step function) are not instantaneous – they have a small slope in the plot. In addition, you need to represent impulses with the height in discrete time equal to the area in continuous time.

When you want to play or plot the discrete-time signal, you need to specify the time increment Ts between samples. As you found in the previous lab, when playing a sound you specify Fs=1/Ts. (Fs is set for you when you load a sound.) When plotting, you need to define a time vector, e.g. t=[0:Ts:end] where end=(length-1)*Ts.

3.1.2 Some Useful MATLAB Commands

- `whos`, list all variables and their sizes.
- `clear`, clears all variables.
- `zeros`, creates a vector (or matrix) of zeros.
- `ones`, creates a vector (or matrix) of ones.
- `conv`, convolves two signals.
- `soundsc`, plays an audio signal, normalizing if the values are greater than +/-1. Requires the sampling rate.

3.1.3 Convolution

MATLAB has a function called `conv(x,h)` that you can use to convolve two discrete-time functions `x(n)` and `h(n)`. It assumes that the time steps are the same in both cases. The input signals must be finite length, and the result of the convolution has a length that is the sum of the lengths of the two signals you are convolving (actually L1+L2-1).

---

[1]This content is available online at <http://cnx.org/content/m14109/1.6/>.

1. Recall that a linear time-invariant system is completely described by its impulse function. In MATLAB, the impulse response must be discrete. For example, consider the system with impulse response

   ```
   h = [1 zeros(1,20) .5 zeros(1,10)];
   ```

   Plot the impulse response using the `plot` command.
2. Consider an input to the system,

   ```
   x = [0 1:10 ones(1,5)*5 zeros(1,40)];
   ```

   Plot the input with the `plot` command.
3. Use the command `conv` to convolve `x` and `h` like this,

   ```
   y = conv(x, h);
   ```

   Use subplot to show the impulse response, input, and output of the convolution. Note that you need to add zeros to the end of x and h (to make them the same length as y) or define a time vector for each signal in order to make the timing comparable in the different subplots.
4. Every non-zero coefficient of the impulse response **h**, acts as an echo. When you convolve the input **x** and impulse response **h**, you add up all the time-shifted and scaled echoes. Try making the second coefficient negative. How does this change the final result?

Exercise 3.1.1 Convolution and Echo

1. Create a new script for this problem. Download the trumpet jazz lick "fall" here[2] , and then load it into MATLAB using `load('fall')` and plot it. Use `whos` to see that the variables `fall` and `Fs` are created for you. (The sampling rate (Fs) for this signal should be 8000 Hz.)
2. Use the following commands to convolve the following impulse response h, with the trumpet sound.

   ```
   Fs = 8000      % for this example
   h = [1 zeros(1,10000) .25 zeros(1,1000)];
   y = conv(fall, h);
   plot(y)
   soundsc(y, Fs)
   ```

3. What if the second echo (in h) is a negative coefficient? When you play it, it should not sound different since your ear is not sensitive to that sort of modification (simple phase change).
4. Now let's build a system that delays the echo for a quarter second by inserting `Fs/4` zeros before the second impulse:

   ```
   h = [1 zeros(1, round(Fs/4)) 0.25 zeros(1,1000)];
   ```

   Pass the `fall` input signal through the system to get the output y:

   ```
   y = conv(h, fall);
   ```

---

[2]http://cnx.org/content/m14109/latest/fall.mat

How do the input and output signals compare in the above step? (Look and listen). Experiment with different numbers of zeros, and try repeating this with some of the built-in MATLAB sounds.

NOTE: Some built-in sounds in MATLAB are `chirp`, `gong`, `handel`, `laughter`, `splat`, and `train`. Load them with the `load` command and the sound data will be loaded into the variable `y` and the sampling rate in `Fs`.

5. Show the TA your script file. You should be able to run it and have it generate any plots and sounds.

NOTE: You can use the `pause` command to pause MATLAB until a key is pressed to prevent it from playing all your sounds at once.

Exercise 3.1.2 Convolution and Smoothing

1. Build a box impulse response:

```
h2=[ones(1,50)/50 zeros(1,20)];
```

Create a new signal `y2` by convolving "fall" with `h2`
2. How does the output sound different from the input signal?
3. Visually, a difference is that the input signal `fall` looks like it's centered around value 0, and the system output `y2` looks like it's more positive. Let's look more closely. Find the average value of the signal `fall` (use `sum(fall)/length(fall)`), and you should see that in fact the `fall` signal isn't really centered around 0.
4. Next, to see what this system does to the input signal, zoom in on part of the signal:

```
subplot(2,1,1), plot(6400:6500, fall(6400:6500))
subplot(2,1,2), plot(6400:6500, y2(6400:6500))
```

The convolved signal should look a little smoother to you. This is because this impulse response applies a low-pass filter to the signal. We'll learn more about filters a bit later, but basically the idea is that the original signal is made up of sounds at many different frequencies, and the lower frequencies pass through the system, but the higher frequencies are attenuated. This affects how it sounds as well as how it looks.

Exercise 3.1.3 Box Function

1. Create a new function called `unitstep.m` in MATLAB. The function should take two parameters, a time vector that specifies the finite range of the signal and a time shift value.

NOTE: Calling `unitstep([time],ts)` should be equivalent to `u(t + ts)`

2. Use the `unitstep` function to create a box-shaped time signal. Write a new function called `boxt.m` that creates a box with specified start and end times `t1` and `t2`. In other words, your function should take three inputs: scalars `t1` and `t2`, and a time vector `t`, and should output a vector of the same size as `t`, which contains the values of `u(t-t1)-u(t-t2)` evaluated at each point in `t`.
3. Create a script file called `boxtscript.m` that uses the function to create a box that starts at time `t = -1` and ends at time `t = 1`, where the signal lasts from time `t = -3` to `t = 3`. Generate three different versions of this box using three different time granularities, where the finest granularity has very sharp edges similar to the ideal box and the coarsest granularity has a step size of 0.5.

NOTE: The different versions should all have the same time span; the difference in the plots should only be at the edges of the box because of artifacts in continuous plotting of a discrete-time signal.

4. Plot all three versions in one figure using subplot and save it as `boxtscript.tif`.

5. Time: If `u` is a vector of length `n` with time span `tu = t1:del:t2`, and `v` is a vector of length `m` with time span `tv = t3:del:t4`, and both have the same time step `del`, then the result of `conv(u,v)` will be a vector of length `n + m - 1` with a time span `tc = (t1+t3):del:(t2+t4)`.

6. Using the box function that you wrote in step 2 with a sufficiently fine grained step size (for example, `del = 0.01`), create box signals from (0,4) and (-1,1), with time span of (-5,10). Find and plot the result of the convolution of the two boxes and save it as `convplot.tif`. Use the above discussion of Time to create the appropriate time vector in your plot. Verify that the timing of signal rising and falling matches what you expect in theory.

7. Amplitude: In the resulting plot from the previous step, you should notice that the amplitude is much higher than the max of 2 that you would expect from analytically computing the convolution. This is because it is thinking that the length of the box is `n` rather `n del`, which impacts the area computation in convolution. To get the correct height, you need to scale by `del`. Scale and plot the resulting function, and verify that the height is now 2. Save the figure as `scaled.tif`

8. Triangle: Design the impulse response for a system `h` and a system input `x` such that you get a perfectly symmetric triangle of length 100 as the system output `y`. Use subplot to plot `x`, `h`, and `y`, and save the plot as `tri.tif`.

9. Be able to demonstrate your code, show your plots, and play sounds.

Chapter 4

Lab 4

4.1 Fourier Series and Gibbs Phenomenon[1]

4.1.1 Introduction

In this lab, we will look at the Fourier series representation of periodic signals using MATLAB. In particular, we will study the truncated Fourier series reconstruction of a periodic function.

4.1.2 Some Useful MATLAB Commands

- `abs`, compute the complex magnitude.
- `angle`, compute the phase angle.
- `clear`, clears all variables.
- `help <command>`, online help.
- `whos`, list all variables and their sizes.

4.1.3 Signal Synthesis

We will see in exercise 3 that we can approximate a square wave with the Fourier series, but first let us approximate something more interesting, say a musical instrument? Many instruments produce very periodic waveforms.

> **Exercise 4.1.1 Synthesizer**
>
> 1. Create a script file called `sigsynth.m` to put your code in for this problem.
> 2. Download the trumpet sound sample `trumpet.mat` from the Sound Resources (Section 2.2) page. The sample rate, `Fs`, of the trumpet is 11,025 Hz. Play this sound with the `sound` command (remember to include the correct sample rate).
> 3. Plot only a small section of the trumpet sound to show three or so periods (try 100 samples or so). Does it looks the same at any time in the sound?
> 4. View the frequency spectrum of this sound by entering the following commands,

```
    Fs = 11025;              % our sample rate is 11025 Hz
Y = fft(trumpet, 512);  % take the fft of trumpet
Ymag = abs(Y);          % take the mag of Y
f = Fs * (0:256)/512;   % get a meaningful axis
plot(f, Ymag(1:257));   % plot Ymag (only half the points are needed)
```

---

[1] This content is available online at <http://cnx.org/content/m13599/1.21/>.

```
xlabel('Frequency (Hz)')
ylabel('Magnitude')
```

You should now see a series of peaks (these are the harmonics of the instrument).

5. We will synthesize the instrument using only the peak information. You can use the "data cursor" tool in MATLAB's figure window to easily read graph data. Write down the frequency and its strength (magnitude) for five to ten of the strongest peaks.

6. Create a function called `addcosines.m` that takes in three vectors: time vector `t`, frequency vector `freq`, and magnitude vector `mag`. Have your new function use a for-loop to add together cosines, one for each frequency/magnitude pair in the `freq` and `mag` vectors. Remember to normalize your output vector after you add up all the cosines (the output should be between -1 and 1), like in the Functions in MATLAB and the Groove Station (Section 2.1) lab. Use the data you collected from the frequency plot of the trumpet sound with your new function to sum cosines at the noted frequencies.

7. Here are some hints for the above. Use a for-loop to create a cosine at each frequency in the freq vector. Your cosine function should look something like this, `mag(i)*cos(2*pi*freq(i)*t);`. Remember your time vector will have the form 0:1/Fs:time_in_seconds.

   NOTE: The command `soundsc` will normalize the input before it plays the sound.

   For example, if you had two harmonics, one at 100 Hz with magnitude 1 and another at 150 Hz with magnitude 2, then your vectors will be,

```
     t = 0:1/Fs:1;  % one second time vector at 11025 Hz
freq = [100 150];
mag = [1 2];
```

8. Play trumpet and your new synthesized sound. Do they sound the same? Use subplot to plot a small section of your new synthesized sound along with the trumpet sound, does it look the same? Save your plot as `synthwaves.tif`.

9. Try synthesizing the sound with fewer frequencies, then try more frequencies. How does this affect the sound of our synthesized trumpet?

10. You will need to show the TA the following files:

```
sigsynth.m
addcosines.m
synthwaves.tif
```

Exercise 4.1.2
That funny phase
You probably noticed in the last problem that even though the wave forms looked fairly different, the sound was similar. Let's look into this a bit deeper with a simpler sound.

1. Create a script file called `phasefun.m` to put your code in for this problem.

2. Pick two harmonic frequencies and generate a signal from two cosines at these frequencies added together and call it `sig1`. Use Fs = 8000 (remember that you can reproduce only frequencies that are less than `Fs/2`).

3. Now generate a second signal called `sig2` exactly the same as the first one, except time delay the second cosine by a half cycle (half of its period).

4. Use subplot to show a few periods of both signals, do they look different? Save the plot as `phasesigs.tif`. What did the time delay do to the phase?
5. Play each signal with `soundsc`, do they sound different?
6. Redo `sig2` with a few different delays and compare the sound to the first signal.
7. Create a `sig3` that is one cosine at some frequency. Now add `sig3` with a timed delayed version of itself and call it `sig4`. Use a quarter cycle delay.
8. Use subplot and plot a few periods of `sig3` and `sig4`. Play them with `soundsc`, do they sound different to you?
9. What is suggested about our hearing capabilities from this experiment?
10. You will need to show the TA the following files:

`phasefun.m`
`phasesigs.tif`

4.1.4 Truncated Fourier Series

In this section, we'll reconstruct the periodic function `x(t)`, shown in Figure 1, by synthesizing a periodic signal from a variable number of Fourier Series coefficients, and observe similarities and differences in the synthesized signal.
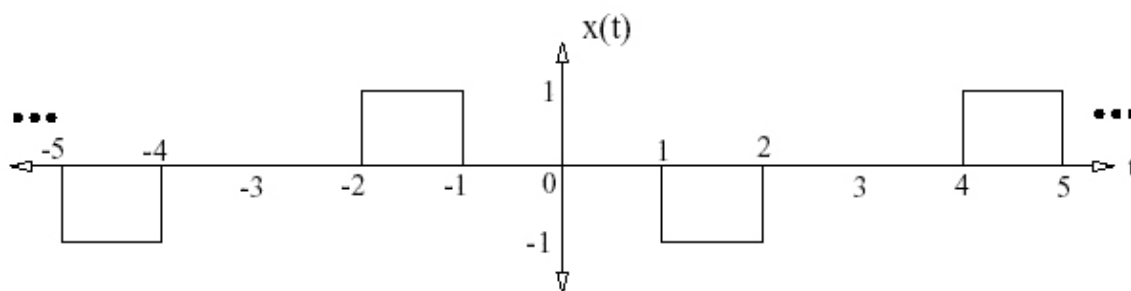

Figure 4.1: Periodic Signal

Exercise 4.1.3 Gibbs phenomena

1. Create a script file called `gibbs.m` to put your code in for this problem.
2. Click here[2] to download the MATLAB function `Ck.m`. Take a look at the contents of the function. This function takes one argument k, and creates the k th Fourier series coefficient for the squarewave above:

$$C_k = \{ \begin{array}{ll} 0 & \text{if } k = 0, \ k \text{ even} \\ \frac{1}{jk\pi}\left[\cos\left(\frac{2k\pi}{3}\right) - \cos\left(\frac{k\pi}{3}\right)\right] & \text{if } k \text{ odd} \end{array} \tag{4.1}$$

[2]http://cnx.org/content/m13599/latest/Ck.m

$C_k(1) = \frac{-1}{j\pi} = 0 + j0.3183$. Plot the magnitude and phase of the coefficients C_k for $k \in \{-10, -9, \ldots, 9, 10\}$. The magnitude and phase should be plotted separately using the subplot command, with the magnitude plotted in the top half of the window and the phase in the bottom half. Place frequency w on the x axis. Use the MATLAB command `stem` instead of `plot` to emphasize that the coefficients are a function of integer-valued (not continuous) k . Label your plots.

3. Save the graph as `Coeff.tif`.

4. Write whatever script/function files you need to implement the calculation of the signal $x(t)$ with a truncated Fourier series:

$$
\begin{aligned}
x(t) &= \sum_{k=-K_{\max}}^{K_{\max}} C_k e^{jk\omega_0 t} \\
     &= \sum_{k=0}^{K_{\max}} 2|C_k| cos(k\omega_0 t + \angle C_k)
\end{aligned}
\tag{4.2}
$$

for a given K_{max}

NOTE: You can avoid numerical problems and ensure a real answer if you use the cosine form. For this example, $w_0 = 1$.

5. Produce plots of $x(t)$ for $t \in [-5, 5]$ for each of the following cases: $K_{max} = 5$; 15; and 30. For all the plots, use as your time values the MATLAB vector `t=-5:.01:5`. Stack the three plots in a single figure using the `subplot` command and include your name in the title of the figure. Save the figure as `FourTrunc.tif`

6. Add clear comments describing what the files do. You will need to show the TA the following files:

```
gibbs.m
Coeff.tif
FourTrunc.tif
```

As you add more cosines you'll note that you do get closer to the square wave (in terms of squared error), but that at the edges there is some undershoot and overshoot that becomes shorter in time, but the magnitude of the undershoot and overshoot stay large. This persistent undershoot and overshoot at edges is called Gibbs Phenomenon.

In general, this kind of "ringing" occurs at discontinuities if you try to synthesize a sharp edge out of too few low frequencies. Or, if you start with a real signal and filter out its higher frequencies, it is "as if" you had synthesized the signal from low frequencies. Thus, low-pass filtering (a filter that only passes low-frequencies) will also cause this kind of ringing.

For example, when compressing an audio signal, higher frequencies are usually removed (that is, the audio signal is low-pass filtered). Then, if there is an impulse edge or "attack" in the music, ringing will occur. However, the ringing (called "pre-echo" in audio) can be heard only before the attack, because the attack masks the ringing that comes after it (this masking effect happens in your head). High-quality MP3 systems put a lot of effort into detecting attacks and processing the signals to avoid pre-echo.

4.1.5 What to Show the TA

Show the TA ALL m-files that you created or edited and the files below.

```
gibbs.m
Coeff.tif
FourTrunc.tif
sigsynth.m
```

```
addcosines.m
synthwaves.tif
phasefun.m
phasesigs.tif
any wav files created
```

4.1.6 Fun Links

An applet here[3] provides a great interface for listening to sinusoids and their harmonics. There are some well-known auditory illusions associated with the perception of pitch here[4] .

---

[3] http://www.phy.ntnu.edu.tw/ntnujava/viewtopic.php?t=33
[4] http://physics.mtsu.edu/∼wmr/julianna.html

Chapter 5

Lab 5

5.1 Filtering Periodic Signals[1]

5.1.1 Introduction

In this lab, we will look at the effect of filtering signals using a frequency domain implementation of an LTI system, i.e., multiplying the Fourier transform of the input signal with the frequency response of the system. In particular, we will filter sound signals, and investigate both low-pass and high-pass filters. Recall that a low-pass filter filters out high frequencies, allowing only the low frequencies to pass through. A high-pass filter does the opposite.

5.1.2 MATLAB Commands and Resources

- `help <command>`, online help for a command.
- `fft`, Fast Fourier Transform.
- `ifft`, Inverse Fourier Transform.
- `sound`, plays sound unscaled (clips input to [-1,1]).
- `soundsc`, plays sound scaled (scales input to [-1,1]).
- `wavread`, reads in WAV file. The sampling rate of the WAV file can also be retrieved, for example, `[x, Fs] = wavread('filename.wav')`, where `x` is the sound vector and `Fs` is the sampling rate.

All of the sounds for this lab can be downloaded from the Sound Resources (Section 2.2) page.

5.1.3 Transforming Signals to the Frequency Domain and Back

When working in MATLAB, the continuous-time Fourier transform cannot be done by the computer exactly, but a digital approximation is done instead. The approximation uses the discrete Fourier transform (more on that in EE 341). There are a couple important differences between the discrete Fourier transforms on the computer and the continuous Fourier transforms you are working with in class: finite frequency range and discrete frequency samples. The frequency range is related to the sampling frequency of the signal. In the example below, where we find the Fourier transform of the "fall" signal, the sampling frequency is `Fs=8000`, so the frequency range is [-4000,4000] Hz (or 2*pi times that for w in radians). The frequency resolution depends on the length of the signal (which is also the length of the frequency representation).

The MATLAB command for finding the Fourier transform of a signal is `fft` (for Fast Fourier Transform (FFT)). In this class, we only need the default version.

---

[1]This content is available online at <http://cnx.org/content/m14481/1.9/>.

```
≫ load fall      %load in the signal
≫ x = fall;
≫ X = fft(x);
```

The `fft` command in MATLAB returns an uncentered result. To view the frequency content in the same way as we are used to seeing it in class, you need to plot only the first half of the result (positive frequencies only) OR use the MATLAB command `fftshift` which toggles between centered and uncentered versions of the frequency domain. The code below will allow you to view the frequency content both ways.

```
     ≫ N = length(x);
≫ pfreq = [0:N/2]*Fs/N;        % index of positive frequencies in fft
≫ Xpos=X(1:N/2+1);             % subset of fft values at positive frequencies
≫ plot(pfreq,abs(Xpos));       % plot magnitude of fft at positive frequencies
≫ figure;
≫ freq = [-(N/2-1):N/2]*Fs/N;  % index of positive AND negative freqs
≫ plot(freq,abs(fftshift(X))); % fftshift actually SWAPS halves of X here. See help.
                    % Convince yourself of why it does this to match up with freq!
```

Note that we are using `abs` in the plot to view the magnitude since the Fourier transform of the signal is complex valued. (Type `X(2)` to see this. Note that X(1) is the DC term, so this will be real valued.)

Try looking at the frequency content of a few other signals. Note that the fall signal happens to have an even length, so N/2 is an integer. If the length is odd, you may have indexing problems, so it is easiest to just omit the last sample, as in `x=x(1:length(x)-1);`.

After you make modifications of a signal in the frequency domain, you typically want to get back to the time domain. The MATLAB command `ifft` will accomplish this task.

```
≫ xnew = real(ifft(X));
```

You need the `real` command because the inverse Fourier transform returns a vector that is complex-valued, since some changes that you make in the frequence domain could result in that. If your changes maintain complex symmetry in the frequency domain, then the imaginary components should be zero (or very close), but you still need to get rid of them if you want to use the `sound` command to listen to your signal.

5.1.4 Low-pass Filtering

An ideal low-pass filter eliminates high frequency components entirely, as in:

$$H_L^{ideal}\left(\omega\right) = \{ \begin{array}{ll} 1 & |\omega| \leq B \\ 0 & |\omega| > B \end{array} \}$$

A real low-pass filter typically has low but non-zero values for $|H_L\left(\omega\right)|$ at high frequencies, and a gradual (rather than an immediate) drop in magnitude as frequency increases. The simplest (and least effective) low-pass filter is given by (e.g. using an RC circuit):

$$H_L\left(\omega\right) = \frac{\alpha}{\alpha + j\omega}, \, \alpha = \text{cutoff frequency.}$$

This low-pass filter can be implemented in MATLAB using what we know about the Fourier transform. Remember that multiplication in the Frequency domain equals convolution in the time domain. If our signal

and filter are both in the frequency domain, we can simply multiply them to produce the result of the system.

$$y(t) = x(t) * h(t)$$

$$Y(\omega) = X(\omega) H(\omega)$$

Below is an example of using MATLAB to perform low-pass filtering on the input signal x with the FFT and the filter definition above.

The cutoff of the low-pass filter is defined by the constant a. The low-pass filter equation above defines the filter H in the frequency domain. Because the definition assumes the filter is centered around w = 0, the vector w is defined as such.

```
   ≫ load fall      %load in the signal
≫ x = fall;
≫ X = fft(x);   % get the Fourier transform (uncentered)

≫ N = length(X);
≫ a = 100*2*pi;
≫ w = (-N/2+1:(N/2))*Fs/N*2*pi;  % centered frequency vector (rad/s)
≫ H = a ./ (a + i*w);            % generate centered sampling of H
≫ plot(w/(2*pi),abs(H))          % w converted back to Hz for plotting
```

The plot will show the form of the frequency response of a system that we are used to looking at, but we need to shift it to match the form that the fft gave us for x.

```
≫ Hshift = fftshift(H);     % uncentered version of H
≫ Y = X .* Hshift';         % filter the signal
```

NOTE: If you are having problems multiplying vectors together, make sure that the vectors are the exact same size. Also, even if two vectors are the same length, they may not be the same size. For example, a row vector and column vector of the same length cannot be multiplied element-wise unless one of the vectors is transposed. The ' operator transposes vectors/matrices in MATLAB.

Now that we have the output of the system in the frequency domain, it must be transformed back to the time domain using the inverse FFT. Play the original and modified sound to see if you can hear a difference. Remember to use the sampling frequency Fs.

```
≫ y = real(ifft(Y));
≫ sound(x, Fs)        % original sound
≫ sound(y, Fs)        % low-pass-filtered sound
```

The filter reduced the signal amplitude, which you can hear when you use the sound command but not with the soundsc which does automatic scaling. Replay the sounds with the soundsc and see what other differences there are in the filtered vs. original signals. What changes could you make to the filter to make a greater difference?

NOTE: Sometimes, you may want to amplify the signal so that it has the same height as the original, e.g., for plotting purposes.

```
≫ y = y * (max(abs(x))/max(abs(y)))
```

Exercise 5.1.1
Low-pass Filtering with Sound
Use `wavread` to load the sound castanets44m.wav. Perform low-pass filtering with the filter defined above, starting with `a = 500*2*pi`, but also try different values.

Play the original and the low-passed version of the sound. Plot their frequency content (Fourier transforms) as well.

Exercise 5.1.2 OPTIONAL: Low-pass Filtering

1. Create an impulse train as the input signal `x(t)` using the following MATLAB command,

```
≫ x = repmat([zeros(1, 99) 1], 1, 5);
```

2. Use the low-pass filter defined earlier to low-pass the impulse train. Choose a cutoff of 20.
3. Plot the two signals `x(t)` and `y(t)` separately using the subplot command. These should be plotted versus the time vector. Label the axes and title each graph appropriately.
4. Look at the plots. Can you explain what is happening to the spike train?

5.1.5 High-pass Filtering

An ideal high-pass filter eliminates low frequency components entirely, as in:

$$H_H^{ideal}(\omega) = \{ \begin{array}{ll} 0 & |\omega| < B \\ 1 & |\omega| \geq B \end{array} \}$$

A real high-pass filter typically has low but non-zero values for $|H_L(\omega)|$ at low frequencies, and a gradual (rather than an immediate) rise in magnitude as frequency increases. The simplest (and least effective) high-pass filter is given by (e.g. using an RC circuit):

$$H_H(\omega) = 1 - H_L(\omega) = 1 - \frac{\alpha}{\alpha + j\omega}, \ \alpha = \text{cutoff frequency.}$$

This filter can be implemented in the same way as the low pass filter above.

Exercise 5.1.3
High-pass Filtering with Sound
The high-pass filter can be implemented in MATLAB much the same way as the low-pass filter. Perform high-pass filtering with the filter defined above on the sound castanets44m.wav. Start with `a = 2000*2*pi`, but also try different values.

Play the original and the high-passed version of the sound. The filtered signal may be to be scaled so that both have the same range on the Y-axis. Plot their frequency responses as well.

5.1.6 Sound Separation

Exercise 5.1.4 Sound Filtering

- Kick'n Retro 235 Inc. recorded a session of a trumpet and drum kit together for their new release. The boss doesn't like the bass drum in the background and wants it out. Unfortunately, there was a malfunction in the mixing board and instead of having two separate tracks for the drums and the trumpet, the sounds mixed together in one track. In order to get this release out on time you will have to use some filtering to eliminate the bass drum from the sound. There is not enough time to bring the drummer and trumpet player back in the studio to rerecord the track.
- Click here to download the mixed.wav sound[2] (`Fs` = 8000 Hz). The mixed sound is created from bassdrum.wav, hatclosed.wav, and shake.mat.
- Try to do something easy but approximate first, and then, if you have more time, see how clean you can get the sound. You may find it helpful to look at the Fourier domain representation of the sounds, but you may not use the individual sounds in your solution.
- Now try to eliminate the trumpet sound, leaving only the drums left in the sound.
- Hint: use high-pass and low-pass filtering.
- Another hint: If you want a more powerful filter, you can try using multiple a/(a+jw) terms in series. Each extra term raises the order of the filter by one and higher order filters have a faster drop-off outside of their passing region.

Exercise 5.1.5 BONUS PROBLEM: Sound Filtering

- Imagine you recorded a trumpet and rainstick together, so that you have the signal, `mixedsig = shake + 10*rainstick`.
- It turns out the producer thinks the rainstick is too new-age and wants it out of the recording. Pretend you do not have the original signals shake or rainstick. Can you take the signal `mixedsig` and process it to get (approximately) only the trumpet sound (shake) out? Try to do something easy but approximate first, and then if you have more time, see how good a reproduction of shake you can get. You may find it helpful to look at Fourier domain of the sounds, but you may not use rainstick.mat or shake.mat in your solution.

---

[2]See the file at <http://cnx.org/content/m14481/latest/mixed.wav>

Chapter 6

Lab 6

6.1 Investigation of Aliasing Effects[1]

6.1.1 Introduction

Aliasing literally means "by a different name" and is used to explain the effect of under-sampling a continuous signal, which causes frequencies to show up as different frequencies. This aliased signal is the signal at a different frequency. This is usually seen as higher frequencies being aliased to lower frequencies. For a 1d signal in time, the aliased frequency components sound lower in pitch. In 2d space, such as images, this can be observed as parallel lines in pinstripe shirts aliasing into large wavy lines. For 2d signals that vary in time, an example of aliasing would be viewing propellers on a plane that seem to be turning slow when they are actually moving at very high speeds.

> NOTE: The Nyquist sampling rate is twice the highest frequency of the signal. This is the minimum rate needed to prevent aliasing.

6.1.2 Signals and Aliasing

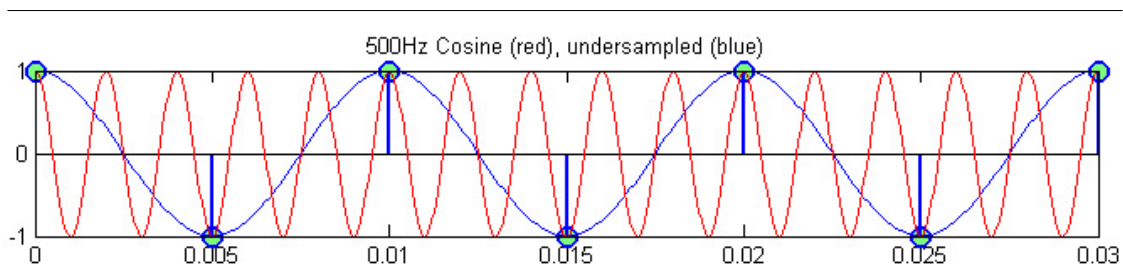In Figure 1 a 500Hz cosine signal is shown in red, and an under-sampled version of the signal in blue.

Figure 6.1: Aliased Signal

Exercise 6.1.1
To see the effects of aliasing on a 1kHz cosine signal create an over-sampled, under-sampled, and critically-sampled version of the signal.

---

[1]This content is available online at <http://cnx.org/content/m13687/1.8/>.

1. Plot a cosine at 1kHz showing at least twenty periods. Use a step size (sampling period) of 1/10kHz. This will be our over-sampled signal. Try playing this signal with `soundsc`. How many samples are needed to make the sound last 2 seconds if the step size is 1/10kHz?

2. Plot the critically-sampled version by applying what you know about Nyquist. Make sure the plot contains at least twenty periods and that you sample at a non-zero point. Listen to this signal with `soundsc`, does it sound the same?

3. Plot the under-sampled version. Make sure the plot contains at least twenty periods. Listen to this signal with `soundsc`, how does it sound now?

4. Plot all three signals stacked on top of each other using subplot. Note that the plot command uses straight line interpolation, so your plots will not look smooth like Figure 1 (which actually uses a much finer sampling period an knowledge of the aliased frequency to generate the smooth undersampled result).

6.1.3 Temporal Aliasing

Have you ever seen an old western movie and noticed that the wagon wheels appear to turn backwards even though the coach is moving forward? This phenomenon is sometimes referred to as the wagon-wheel effect, but is really an effect of temporal aliasing. You can see the same effect easily on anything with a spoked wheel, such as wheels on a stage coach and airplane propellers.

Wagon-wheels, stage coaches, horses, and airplane propellers?? What's this have to do with signal processing? Actually, quite a lot, not the wagon-wheels directly, but how the images of the wagon-wheels are captured. The video you watch from a movie or tv show is actually sampled in time (hence temporal). Typically a movie is captured at 24 frames per second (FPS).

Exercise 6.1.2
Now it's your turn to be the cinematographer. For this problem you will take an image of a wagon-wheel and "capture" a MATLAB movie at different frame rates of the wheel rotating. After the movie is made, you will be able to play it back, and if everything worked, be able to see the wheel spin.

A movie of a rotating wheel is a signal in time, and at each instant in time, instead of just one point (like a normal x(t) signal), you have a whole image defined. Thus, if you have an image of an arrow rotating, Figure 2, where the image rotates ten times per second, then the period is 1/10 second, because every 1/10 second the image (signal) is at the same value again. Thus image(t+n/10) = image(t) for all integers n.

frame 1   frame 2   frame 3   frame 4   frame 5   frame 6
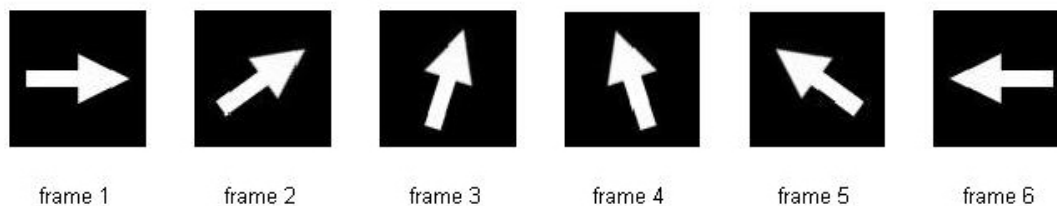
Figure 6.2: Frames of rotating arrow.

If an image rotates at 10 Hz (10 rotations per second), then what is the Nyquist sampling rate so that you can reconstruct the temporal signal? Recall that the signal will be critically sampled when using a sampling rate that is twice the highest frequency in the signal (20 Hz, in this case). Anything above that will be over-sampled, and fewer samples/second will be under-sampled.

Check your understanding: standard film is captured at 24 frames per second. What's the highest frequency of motion that can be reconstructed without aliasing?

Create three movies to show the wheel being over-sampled (appears to be rotating clockwise), under-sampled (appears to be rotating counter-clockwise), and critically-sampled (appears stationary). In each case rotate the wheel at the same rate and only change the frame rate in the `movie2avi` command (keep the FPS under 30).

Write a Matlab function named `wheel.m` to create a movie showing the spokes image (download it here[2]) rotate clockwise at a constant speed. The function should take parameters to change the frame rate and the speed of the rotation. Save the movies as `wheel-oversample.avi`, `wheel-undersample.avi`, and `wheel-critsample.avi`. Label the plot with the frame rate used for each of the movies and the degrees per frame. Here some tips below to help you get started.

- You will need to use the following Matlab commands: `imread`, `imshow`, `imrotate`, `getframe`, and `movie2avi`.

  NOTE: Passing a negative angle in the `imrotate` command rotates clockwise, and a positive angle rotates counterclockwise.

  Another useful command you can use to help formatting labels for the figure is `sprintf`. For more information use the help system in Matlab.
- Use `myImageRotated = imrotate(myImage, theta, 'bilinear', 'crop')` for the rotate command.
- One way to do this is rotate the image by a number of degrees for each frame. The angle can be split into two variables; `degPerFrame` will be our speed and `theta` will be the actual number of degrees to rotate for the rotate command. Remember to change `degPerFrame` to reflect the same speed when changing the frame rate. Now we can setup a `for` loop something like this,

```
for i = 1:FPS*TIME
    % rotate the image

    % display the image

    % label the plot showing the FPS and speed of the wheel

    pause(0.01)                   % allows time for the plot to draw
    myMovie(i) = getframe(gcf);   % Capture the frame
    theta = theta + degPerFrame;  % Calculate the angle for next frame
end

% save the avi file
```

- Can you use `degPerFrame` to relate to degrees per second? Given some frame rate, how many degrees pass each frame to make a rotation of 360° take 1 second? At a given frame rate, can you calculate the number of frames are needed to last a given amount of time, say 3 seconds?.
- Once your `for` loop is done, you will need to save the movie as an avi to watch it. Use the `movie2avi` function to save the movie. Why can't we just watch the wheel as it is drawing in the `for` loop?

Now try the same problem with a different picture of your choice. Can you get it to appear to move backwards? Save the movie as `myMovie.avi`.

Show the TA the following files:

---

[2]http://cnx.org/content/m13687/latest/spokes.tif

```
wheel.m
wheel-oversample.avi
wheel-undersample.avi
wheel-critsample.avi
myMovie.avi
```

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

Attributions

Collection: *Continuous Time Linear Systems Laboratory (EE 235)*
Edited by: University Of Washington Dept. of Electrical Engineering
URL: http://cnx.org/content/col10374/1.8/
License: http://creativecommons.org/licenses/by/2.0/

Module: "Introduction to MATLAB and Scripts"
By: University Of Washington Dept. of Electrical Engineering
URL: http://cnx.org/content/m13554/1.18/
Pages: 1-5
Copyright: University Of Washington Dept. of Electrical Engineering
License: http://creativecommons.org/licenses/by/2.0/

Module: "Functions in MATLAB and the Groove Station"
By: University Of Washington Dept. of Electrical Engineering
URL: http://cnx.org/content/m13555/1.25/
Pages: 7-10
Copyright: University Of Washington Dept. of Electrical Engineering
License: http://creativecommons.org/licenses/by/2.0/

Module: "Sound resources"
By: UW EE235 TA UW EE235 TA
URL: http://cnx.org/content/m13854/1.7/
Page: 10
Copyright: UW EE235 TA UW EE235 TA
License: http://creativecommons.org/licenses/by/2.0/

Module: "Convolution"
By: UW EE235 TA UW EE235 TA
URL: http://cnx.org/content/m14109/1.6/
Pages: 11-14
Copyright: UW EE235 TA UW EE235 TA
License: http://creativecommons.org/licenses/by/2.0/

Module: "Fourier Series and Gibbs Phenomenon"
By: University Of Washington Dept. of Electrical Engineering
URL: http://cnx.org/content/m13599/1.21/
Pages: 15-19
Copyright: University Of Washington Dept. of Electrical Engineering
License: http://creativecommons.org/licenses/by/2.0/

Module: "Filtering Periodic Signals"
By: UW EE235 TA UW EE235 TA
URL: http://cnx.org/content/m14481/1.9/
Pages: 21-25
Copyright: UW EE235 TA UW EE235 TA
License: http://creativecommons.org/licenses/by/2.0/

Module: "Investigation of Aliasing Effects"
By: University Of Washington Dept. of Electrical Engineering
URL: http://cnx.org/content/m13687/1.8/
Pages: 27-30
Copyright: University Of Washington Dept. of Electrical Engineering
License: http://creativecommons.org/licenses/by/2.0/

Continuous Time Linear Systems Laboratory (EE 235)
Introduction to continuous time signal analysis. Basic signals including impulses, pulses, and unit steps. Periodic signals. Convolution of signals. Fourier series and transforms in discrete and continuous time. Computer laboratory. This development of these labs was supported by the National Science Foundation under Grant No. DUE-0511635. Any opinions, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

About Connexions
Since 1999, Connexions has been pioneering a global system where anyone can create course materials and make them fully accessible and easily reusable free of charge. We are a Web-based authoring, teaching and learning environment open to anyone interested in education, including students, teachers, professors and lifelong learners. We connect ideas and facilitate educational communities.

Connexions's modular, interactive courses are in use worldwide by universities, community colleges, K-12 schools, distance learners, and lifelong learners. Connexions materials are in many languages, including English, Spanish, Chinese, Japanese, Italian, Vietnamese, French, Portuguese, and Thai. Connexions is part of an exciting new information distribution system that allows for **Print on Demand Books**. Connexions has partnered with innovative on-demand publisher QOOP to accelerate the delivery of printed course materials and textbooks into classrooms worldwide at lower prices than traditional academic publishers.