Laboratory Manual

TO ACCOMPANY

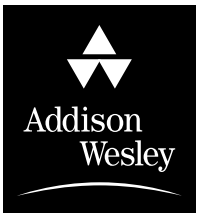
STRUCTURE AND INTERPRETATION OF

Signals and Systems

Edward A. Lee

Pravin Varaiya

University of California at Berkeley



Boston San Francisco New York
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

Reproduced by Addison Wesley from electronic files supplied by Windfall Software.

ISBN: 0-321-16876-3

Laboratory Manual

This laboratory manual contains laboratory exercises based on MATLAB and Simulink.* The purpose of these exercises is to help reconcile the declarative (what is) and imperative (how to) points of view on signals and systems. The mathematical treatment that dominates in the associated text is declarative in that it asserts properties of signals and studies the relationships between signals that are implied by systems. This laboratory manual focuses on an imperative style, where signals and systems are constructed procedurally.

MATLAB and Simulink, distributed by The MathWorks, Inc., are chosen as the basis for these exercises because they are widely used by practitioners in the field, and because they are capable of realizing interesting systems. Why use both MATLAB and Simulink? Although they are integrated into a single package, MATLAB and Simulink are two very different pieces of software with radically different approaches to modeling of signals and systems. MATLAB is an imperative programming language, whereas Simulink is a block diagram language. In MATLAB, one specifies the sequence of steps that construct a signal or operate on a signal to produce a new signal. In Simulink, one connects blocks that implement elementary systems to construct more interesting systems. The systems we construct are aggregates of simpler systems.

MATLAB fundamentally operates on matrices and vectors. Simulink fundamentally operates on discrete and continuous-time signals. Finite discrete-time

* MATLAB and Simulink are registered trademarks of The MathWorks, Inc.

signals, of course, can be represented as vectors. Continuous-time signals, however, can only be approximated. Simulink, since it is a computer program, must of course approximate continuous-time signals by discretizing time. But that approximation is largely transparent, and the user (the model builder) can pretend that he or she is operating directly on continuous-time signals.

There is considerable value in becoming adept with these software packages. MATLAB and Simulink are often used in practice for "quick-and-dirty" prototyping of concepts. In a matter of a few hours, very elaborate models can be constructed. This contrasts with the weeks or months that would often be required to build a hardware prototype to test the same concept.

Of course, a conventional programming language such as C++ or Java could also be used to construct prototypes of systems. However, these languages lack the rich libraries of built-in functions that MATLAB and Simulink have. A task as conceptually simple as plotting a waveform can take weeks of programming in Java to accomplish well. Algorithms, such as the FFT or filtering algorithms, are also built in, saving considerable effort.

MATLAB and Simulink both have capabilities that are much more sophisticated than anything covered in this text. This may be a bit intimidating at first ("what the heck is singular-value decomposition!?"). In fact, these tools are rich enough in functionality to keep you busy for an entire career in engineering. You will need to learn to ignore what you don't understand, and focus on building up your abilities gradually.

If you have no background in programming, these exercises will be difficult at first. MATLAB, at its root, is a fairly conventional programming language, and it requires a clear understanding of programming concepts such as variables and flow of control (for loops, while loops). As programming languages go, it is an especially easy one to learn. Its syntax (the way commands are written) is straightforward and close to that of the mathematical concepts that it emulates. Moreover, since it is an interpreted language (in contrast to a compiled language), you can easily experiment by just typing in commands at the console and seeing what happens. Be fearless! The worst that can happen is that you will have to start over.

These labs assume the computer platform is Microsoft Windows, although any platform capable of running MATLAB and Simulink will work, as long as it has full support for sound and images.

Mechanics of the labs

The labs are divided into two distinct sections, **in-lab** and **independent**. The purpose of the in-lab section is to introduce concepts needed for later parts of the lab. Each in-lab section is designed to be completed during a scheduled lab time with an instructor present to clear up any confusing or unclear concepts. The in-lab section is completed by obtaining the signature of an instructor on a verification sheet.

The independent section begins where the in-lab section leaves off. It can be completed within a scheduled lab period, or may be completed independently. Students should write a brief summary of their solutions to the lab exercise. The summary should clearly answer each question posed in the independent section of the lab.

The lab writeup should be kept simple. It will typically include the names of the members of the group (if the lab is done by a group), the time of the lab section, the name of the lab, and the date. It should then proceed to give clear answers to each of the questions posed by the lab. MATLAB code should be provided in a fixed-width font (Courier New, 10pt, for example) and plots should be clearly labeled and referenced in the writeup. Plots may be included directly in the flow of the analysis. If included on a separate page, two to eight plots should be placed on the same page, depending on the nature of the plots. Students can copy MATLAB plots into most word processors using the Copy Figure command in the Edit menu.

Here is an example of a response to a portion of a lab:

2. Simple Low Pass Filter

Figure L.1 shows the data before (top) and after (bottom) the low pass filter. The low pass filter has the effect of smoothing sharp transitions in the original.

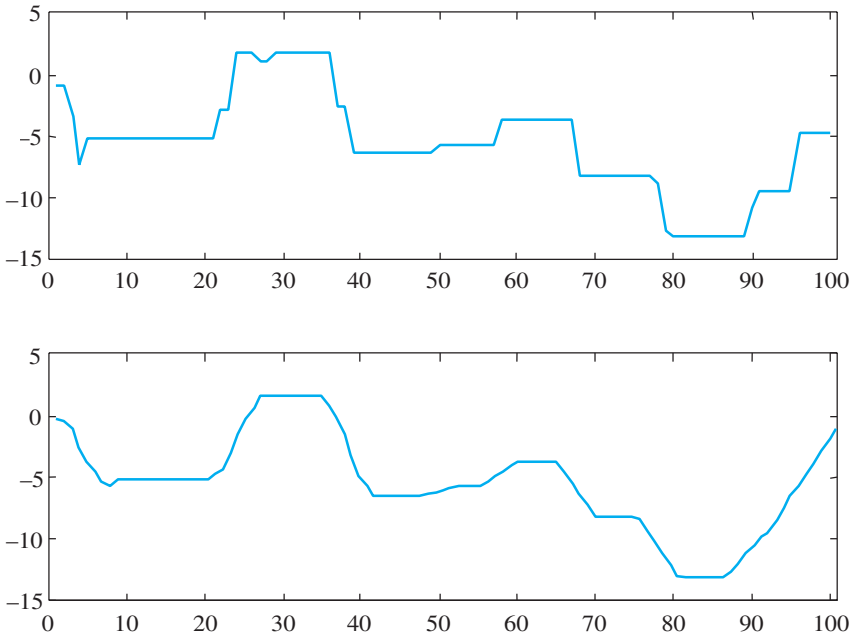


FIGURE L.1: Before and after LPF.

For instance, notice the disappearance of the step from sample points 91 to 94. The MATLAB code used to generate the smoothed waveform v1 from the original waveform x1 is:

```
h5 = [1 1 1 1 1] / 5;
v1 = firfilt(x1, h5);
```

Notes to the instructor

How to schedule the labs depends somewhat on the specific organization of a course. At Berkeley, we have a 15 week course that covers chapters 1 through 11, and we organize the labs as follows. In the first week, there is no lab assignment. In the second week, the lab meeting is devoted to instructional technology. It is intended to ensure that all participants in the class are comfortable with the computing environment and software. The specific tasks that we ask students to accomplish in this first meeting are:

- Log in to a computer.
- Find and print the instructor verification sheet for this lab.
- Access the class Web page, and find and run at least one sound applet.
- Send e-mail to the lab teaching assistant.
- Start MATLAB and access its on-line help and its on-line demos. Find the "help desk" and spend some time in the section "getting started." Get familiar with the tool.
- Start Simulink and access its on-line demos.

In weeks 3 through 15, students complete one lab per week, with one gap where the scheduled lab time is used to review for the midterm exam. The lectures and reading assignments are closely coordinated with the lab assignments so that students have seen in lecture the required background material prior to the lab meeting.

L.1 Arrays and sound

The purpose of this lab is to explore arrays in MATLAB and to use them to construct sound signals. The lab is designed to help you become familiar with the fundamentals of MATLAB. It is self-contained in the sense that no additional documentation for MATLAB is needed. Instead, we rely on the online help facilities. Some people, however, much prefer to sit down with a tutorial text about a piece of software, rather than relying on online help. There are many excellent books that introduce MATLAB. Check your local bookstore or The MathWorks' Web site (http://www.mathworks.com/).

Note that there is some potential confusion because MATLAB uses the term "function" somewhat more loosely than we do when we refer to mathematical functions. Any MATLAB command that takes arguments in parentheses is called a function. And most have a well-defined domain and range, and do, in fact, define a mapping from the domain to the range. These can be viewed formally as a (mathematical) function. Some, however, such as `plot` and `sound` are a bit harder to view this way. The last exercise here explores this relationship.

L.1.1 In-lab section

To run MATLAB simply double-click on the MATLAB icon on the desktop, or find the MATLAB command in the start menu. This will open a MATLAB command window, which displays a prompt ">>". You type commands at the prompt. The ">>" is the MATLAB prompt. You do not type that part. Explore the built-in demos by typing `demo`.

MATLAB provides an online help system accessible by using the `help` command. For example, to get information about the function `size`, enter the following:

```
>> help size
```

There also is a help desk (formatted in HTML for viewing from a Web browser) with useful introductory material. It is accessed from the Help menu. If you have no prior experience with MATLAB, see the topic "Getting Started" in the help desk. Spend some time with this. You can find in the help desk all the information you need to carry out the following exercises.

1. A variable in MATLAB is an array. An array has dimensions $N \times M$, where N and M are in *Naturals*. N is the number of **rows** and M is the number of **columns**. If $N = M = 1$, the variable is a **scalar**. If $N = 1$ and $M > 1$, then the variable is a **row vector**. If $N > 1$ and $M = 1$, then the variable is a **column vector**. If both N and M are greater than one, then the variable is a **matrix**, and if $N = M$, then the variable is a **square matrix**. The coefficients of an array are real or complex numbers.

   (a) Each of the following is an assignment of a value to a variable called `array`. For each, identify the dimensions of the array (M and N), and

identify whether the variable is a scalar, row vector, column vector, or matrix.

```
array = [1 2 3 4 5]
array = [1:5]
array = 1:5
array = [1:1:5]
array = [1:-1:-5]
array = [1 2; 3 4]
array = [1; 2; 3; 4]
```

(b) Create a 2×3 matrix containing arbitrary data. Explore using the MATLAB functions `zeros`, `ones`, `eye`, and `rand` to create the matrix. Find a way to use the square matrix `eye(2)` as part of your 2×3 matrix. Verify the sizes of your arrays using `size`.

(c) Use the MATLAB commands `size` and `length` to determine the length of the arrays given by `1:0.3:10` and `1:1:-1`. Consider more generally the array constructor pattern

```
array = start : step : stop
```

where `start`, `stop`, and `step` are scalar variables or real numbers. How many elements are there in `array`? Give an expression in MATLAB in terms of the variables `start`, `stop`, and `step`. That is, we should be able to do the following:

```
>> start = 1;
>> stop = 5;
>> step = 1;
>> array = start:step:stop;
```

and then evaluate your expression and have it equal `length(array)`. (Notice that the semicolons at the end of each command above suppress MATLAB's response to each command.) Hint: To get a general expression, you will need something like the `floor` function. Verify your answer for the arrays `1:0.3:10` and `1:1:-1`.

2. MATLAB can be used as a general-purpose programming language. Unlike a general-purpose programming language, however, it has special features for operating on arrays that make it especially convenient for modeling signals and systems.

(a) In this exercise, we will use MATLAB to compute

$$\sum_{k=0}^{25} k.$$

Use a `for` loop (try `help for`) to specify each individual addition in the summation.

(b) Use the `sum` function to give a more compact, one-line specification of the sum in part (a). The difference between these two approaches illustrates the difference between using MATLAB and using a more traditional programming language. The `for` loop is closer to the style one would use with C++ or Java. The `sum` function illustrates what MATLAB does best: compact operations on entire arrays.

(c) In MATLAB, any built-in function that operates on a scalar can also operate on an array. For example,

```
>> sin(pi/4)

ans =

    0.7071

>> sin([0 pi/4 pi/2 3*pi/4 pi])

ans =

         0    0.7071    1.0000    0.7071    0.0000
```

This feature is called **vectorization**. Use vectorization to construct a vector that tabulates the values of the sin function for the set $\{0, \pi/10, 2\pi/10, \ldots, \pi\}$. Use the colon notation explored in the previous exercise.

(d) Given two arrays `A` and `B` that have the same dimensions, MATLAB can multiply the elements pointwise using the `.*` operator. For example,

```
>> [1 2 3 4].*[1 2 3 4]

ans =

     1     4     9    16
```

Use this pointwise multiply to tabulate the values of sin^2 for the set

$$\{0, \pi/10, 2\pi/10, \ldots, \pi\}.$$

3. A discrete-time signal may be approximated in MATLAB by a vector (either a row or a column vector). In this exercise, you build a few such vectors and plot them.

(a) Create an array that is a row vector of length 36, with zeros everywhere except in the 18th position, which has value 1. (Hint: Try `help zeros` to find a way to create a row vector with just zeros, and then assign the 18th element of this vector the value one.) This array approximates a discrete-time **impulse**, which is a signal that is zero everywhere except at one sample point. We will use impulses to study linear systems. Plot the impulse signal, using both `plot` and `stem`.

(b) Sketch by hand the sine wave $x : [-1, 1] \rightarrow \mathit{Reals}$, given by

$$\forall\, t \in [-1, 1], \quad x(t) = \sin(2\pi \times 5t + \pi/6).$$

In your sketch carefully plot the value at time 0. Assume the domain represents time in seconds. What is the frequency of this sine wave in Hertz and in radians per second, what is its period in seconds, and how many complete cycles are there in the interval $[-1, 1]$?

(c) Sample the function x from the previous part at 8 kHz, and using MATLAB, plot the samples for the entire interval $[-1, 1]$. How many samples are there?

(d) Change the frequency of the sine wave from the previous section to 440 Hz and plot the signal for the interval $[-1, 1]$. Why is the plot hard to read? Plot the samples that lie in the interval $[0, 0.01]$ instead (this is a 10-msec interval).

(e) 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 PC. The second argument specifies the sampling frequency in Hertz. The values in `sampledSignal` are assumed to be real numbers in the range $[-1.0, 1.0]$. Values outside this range are clipped to -1.0 or 1.0. Use this function to listen to the signal you created in the previous part. Listen to both a 10-msec interval and 2-second interval. Describe what you hear.

(f) Listen to

```
>> sound(0.5*sampledSignal,frequency)
```

and

```
>> sound(2*sampledSignal,frequency)
```

where `sampledSignal` is the signal you created in part (d) above. Explain in what way are these different from what you heard in the previous part. Listen to

```
>> sound(sampledSignal,frequency/2)
```

and

```
>> sound(sampledSignal,frequency*2)
```

Explain how these are different.

L.1.2 *Independent section*

1. Use MATLAB to plot the following continuous-time functions f: $[-0.1, 0.1] \rightarrow$ *Reals*:

$$\forall\, t \in [-0.1, 0.1], \quad f(t) = \sin(2\pi \times 100t)$$

$$\forall\, t \in [-0.1, 0.1], \quad f(t) = \exp(-10t)\sin(2\pi \times 100t)$$

$$\forall\, t \in [-0.1, 0.1], \quad f(t) = \exp(10t)\sin(2\pi \times 100t)$$

The first of these is a familiar sinusoidal signal. The second is a sinusoidal signal with a decaying exponential envelope. The third is a sinusoidal signal with a growing exponential envelope. Choose a sampling period so that the plots closely resemble the continuous-time functions. Explain your choice of the sampling period. Use `subplot` to plot all three functions in one tiled figure. Include the figure in your lab report.

2. Use MATLAB to listen to a one-second sinusoidal waveform scaled by a decaying exponential given by

$$\forall\, t \in [0, 1], \quad f(t) = \exp(-5t)\sin(2\pi \times 440t).$$

Use a sample rate of 8,000 samples/second. Describe how this sound is different from sinusoidal sounds that you listened to in the in-lab section.

3. Construct a sound signal that consists of a sequence of half-second sinusoids with exponentially decaying envelopes, as in the previous part, but with a sequence of frequencies: 494, 440, 392, 440, 494, 494, and 494. Listen to the sound. Can you identify the tune? In your lab report, give the MATLAB commands that produce the sound.
4. This exercise explores the relationship between MATLAB functions and mathematical functions.

   (a) The `sound` function in MATLAB returns no value, as you can see from the following:

```
>> x = sound(n)
??? Error using ==> sound
Too many output arguments.
```

   Nonetheless, `sound` can be viewed as a function, with its range being the set of sounds. Read the help information on the `sound` function carefully and give a precise characterization of it as a mathematical function (define its domain and range). You may assume that the elements of MATLAB vectors are members of the set *Doubles*, double-precision floating-point numbers, and you may, for simplicity, consider only the one-argument version of the function, and model only monophonic (not stereo) sound.

   (b) Give a similar characterization of the `soundsc` MATLAB function, again considering only the one-argument version and monophonic sound.

   (c) Give a similar characterization of the `plot` MATLAB function, considering the one-argument version with a vector argument.

Instructor Verification Sheet for Lab L.1

Name: _____ **Date:** _____

1. MATLAB arrays.

   Instructor verification: _____

2. MATLAB programming.

   Instructor verification: _____

3. Discrete-time signals in MATLAB.

   Instructor verification: _____

L.2 Images

The purpose of this lab is to explore images and colormaps. You will create synthetic images and movies, and you will process a natural image by blurring it and by detecting its edges.

L.2.1 Images in MATLAB

Figure L.2 shows a grayscale image where the intensity of the image varies sinusoidally in the vertical direction. The top row of pixels in the image is white. As you move down the image, it gradually changes to black, and then back to white, completing one cycle. The image is 200 × 200 pixels so the vertical frequency is 1/200 cycles per pixel. The image rendered on the page is about 10 × 10 centimeters, so the vertical frequency is 0.1 cycles per centimeter. The image is constant horizontally (it has a horizontal frequency of 0 cycles per pixel).

We begin this lab by constructing the MATLAB commands that generate this image. To do this, you need to know a little about how MATLAB represents images. In fact, MATLAB is quite versatile with images, and we will only explore a portion of what it can do.

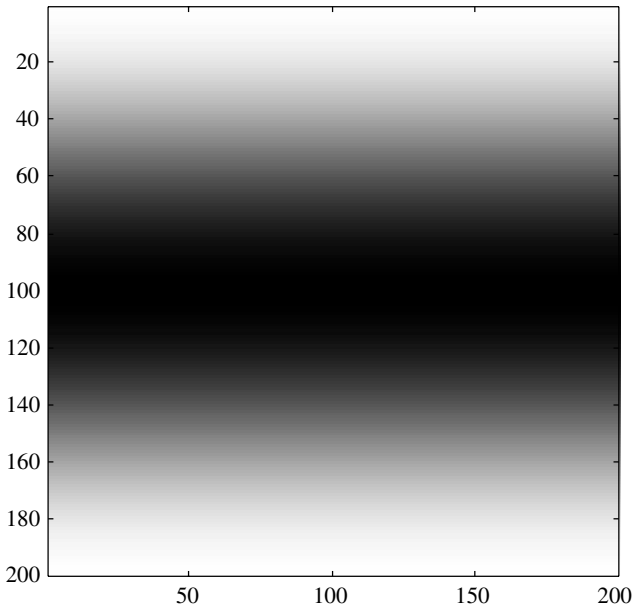


FIGURE L.2: An image where the intensity varies sinusoidally in the vertical direction.

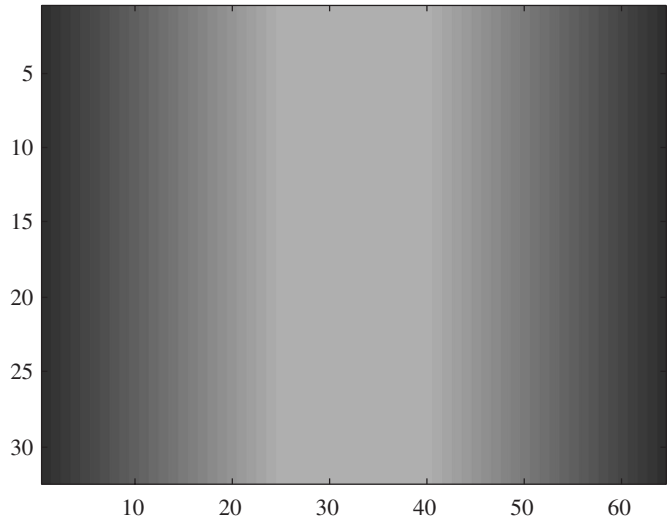


FIGURE L.3: An image rendered by MATLAB.

An image in MATLAB can be represented as an array with two dimensions (a matrix) where each element of the matrix indexes a colormap. Consider, for example, the image constructed by the `image` command:

```
>> v = [1:64];
>> image(v);
```

This should create an image something like that shown in figure L.3, but in color.

The image is 1 pixel high by 64 pixels wide (MATLAB, by default, stretches the image to fit the standard rectangular graphic window, so the one pixel vertically is rendered as a very tall pixel). You could use the `repmat` MATLAB function to make an image taller than 1 pixel by just repeating this row some number of times.

The pixels each have value ranging from 1 to 64. These index the default colormap, which has length 64 and colors ranging from blue to red through the rainbow. To see the default colormap numerically, type

```
>> map = colormap
```

To verify its size, type

```
>> size(map)

ans =

    64     3
```

Notice that it has 64 rows and three columns. Each row is one entry in the colormap. The three columns give the amounts of red, green, and blue, respectively, in the colormap. These amounts range from 0 (none of the color present) to 1.0 (the maximum amount of the color possible). Examine the colormap to convince yourself that it begins with blue and ends with red.

Change the colormap using the `colormap` command as follows:

```
>> map = gray(256);
>> colormap(map);
>> image([1:256]);
```

Examine the `map` variable to understand the resulting image. This is called a **grayscale colormap**.

L.2.2 *In-lab section*

1. What is the representation in a MATLAB colormap for the color white? What about black?
2. Create a 200 × 200 pixel image like that shown in figure L.2. You will want the colormap set to `gray(256)`, as indicated above. Note that when you display this image using the `image` command, it probably will not be square. This is because of the (somewhat annoying) stretching that MATLAB insists on doing to make the image fit the default graphics window. To disable the stretching and get a square image, issue the command `axis image`:

   ```
   axis image
   ```

   As usual with MATLAB, a brute-force way to create matrices is to use for loops, but there is almost always a more elegant (and faster) way that exploits MATLAB's ability to operate on arrays all at once. Try to avoid using for loops to solve this and subsequent problems.
3. Change your image so that the sinusoidal variations are horizontal rather than vertical. Vary the frequency so that you get four cycles of the sinusoid instead of one. What is the frequency of this image?
4. An image can have both vertical and horizontal frequency content at the same time. Change your image so that the intensity at any point is the product of a vertical and horizontal sinusoid. Be careful to stay within the numerical range that indexes the colormap.
5. Get the image file from

   ```
   http://www.aw.com/lee_varaiya/images/helen.jpg
   ```

   Save it in some directory where you have write permission with the name "helen.jpg".

   In MATLAB, change the current working directory to that directory using the `cd` command. Then use `imfinfo` to get information about the file, as follows:

```
>> imfinfo('helen.jpg')

ans =

           Filename: 'helen.jpg'
        FileModDate: '27-Jan-2000 10:48:16'
           FileSize: 18026
             Format: 'jpg'
      FormatVersion: ''
              Width: 200
             Height: 300
           BitDepth: 24
          ColorType: 'truecolor'
    FormatSignature: ''
```

Make a note of the file size, which is given in bytes. Then use `imread` to read the image into MATLAB and display it as follows:

```
>> helen = imread('helen.jpg');
>> image(helen);
>> axis image
```

Use the `whos` command to identify the size, in bytes, and the dimensions of the `helen` array. Can you infer from this what is meant by `truecolor` above? The file is stored in JPEG format, where JPEG, which stands for Joint Pictures Expert Group, is an image representation standard. The `imread` function in MATLAB decodes JPEG images. What is the compression ratio achieved by the JPEG file format (the compression ratio is defined to be size of the uncompressed data in bytes divided by the size of the compressed data in bytes)?

6. The `helen` array returned by `imread` has elements that are of type `uint8`, which means unsigned 8-bit integers. The possible values for such numbers are the integers from 0 to 255. The upper left pixel of the image can be accessed as follows:

```
>> pixel = helen(1,1,:)

pixel(:,:,1) =

   205


pixel(:,:,2) =

   205
```

```
pixel(:,:,3) =

   205
```

In this command, the final argument is ":", which means to MATLAB, return all elements in the third dimension. The information about the result is:

```
>> whos pixel
  Name        Size          Bytes  Class

  pixel       1x1x3             3  uint8 array

Grand total is 3 elements using 3 bytes
```

MATLAB provides the `squeeze` command to remove dimensions of length one:

```
>> rgb = squeeze(pixel)

rgb =

   205
   205
   205
```

Find the RGB values of the lower right pixel. By looking at the image, and correlating what you see with these RGB values, infer how white and black are represented in truecolor images.

L.2.3 *Independent section*

1. Construct a mathematical model for the MATLAB `image` function as used in parts 3 and 4 of the in-lab section by giving its domain and its range. Notice that the colormap, although it is not passed to `image` as an argument, is in fact an argument. It is passed in the form of a **global variable**, the current colormap. Your mathematical model should show this as an explicit argument.
2. In MATLAB, you can create a movie using the following template:

```
numFrames = 15;
m = moviein(numFrames);
for frame = 1:numFrames;
   ... create an image X ...
   image(X), axis image
   m(:,frame) = getframe;
end
movie(m)
```

The line with the `getframe` command grabs the current image and makes it a frame of the movie. Use this template to create a vertical sinusoidal image where the sine waves appear to be moving upward, like waves in water viewed from above (i.e., create something like figure L.2, but where the wave appears to be moving upward). Try `help movie` to learn about various ways to display this movie.

3. We can examine individually the contributions of red, green, and blue to the image by creating **color separations**. MATLAB makes this very easy for truecolor images by providing its versatile array indexing mechanism. To extract the red portion of the `helen` image created previously, we can simply indicate:

```
red = helen(:,:,1);
```

The result is a 300×200 array of unsigned 8-bit integers, as we can see from the following:

```
>> whos red
  Name      Size          Bytes  Class

  red     300x200         60000  uint8 array

Grand total is 60000 elements using 60000 bytes
```

(Note that, strangely, the `squeeze` command is not needed whenever the last dimension(s) collapse to size 1.) If we display this array, its value will be interepreted as indexes into the current color map:

```
image(red), axis image
```

If the current colormap is the default one, then the image will look very off indeed (and very colorful). Change the colormap to grayscale to get a more meaningful image:

```
map = gray(256);
colormap(map);
```

The resulting image gives the red portion of the image, albeit rendered in black and white. Construct a colormap to render it in red. Show the MATLAB code that does this in your report (you need not show the image). Then give similar color separations for the green and blue portions. Again, showing the MATLAB code is sufficient. Hint: Create a matrix to multiply pointwise by the `map` matrix above (using the `.*` operator) to zero out two of its three columns. The `zeros` and `ones` functions might be useful.

4. A moving average can be applied to an image, with the effect of blurring it. For simplicity, operate on a black-and-white image constructed from the preceding red color separation as follows:

```
>> bwImage = double(red);
>> image(bwImage), axis image
>> colormap(gray(256))
```

The first line converts the image to an array of doubles instead of unsigned 8-bit integers because MATLAB cannot operate numerically on unsigned 8-bit integers. The remaining two lines simply display the image using a grayscale colormap.

Construct a new image where each pixel is the average of 25 pixels in the original image, where the 25 pixels lie in a 5×5 square. The new image will need to be slightly smaller than the original (figure out why). The result should be a blurred image because the moving average reduces the high frequency content of a signal, and sharp edges are high-frequency phenomena.

5. A simple way to perform edge detection on a black-and-white image is to calculate the difference between a pixel and the pixel immediately above it and to the left of it. If either difference exceeds some threshold, we decide there is an edge at that position in the image. Perform this calculation on the image `bwImage` given in the previous part. To display with the edges, start with a white image the same size or slightly smaller than the original image. When you detect an edge at a pixel, replace the white pixel with a black one. The resulting image should resemble a line drawing of Helen. Experiment with various threshold values. Hint: To perform the threshold test, you will probably need the MATLAB `if` command. Try `help if` and `help relop`.

   Note: Edge detection is often the first step in **image understanding**, which is the automatic interpretation of images. A common application of image understanding is **optical character recognition** or **OCR**, which is the transcription of printed documents into computer documents.

   The difference between pixels tends to emphasize high-frequency content in the image and deemphasize low-frequency content. This is why it is useful in detecting edges, which are high-frequency content. This is obvious if we note that frequency in images refers to the rate of change of intensity over space. That rate is very fast at edges.

Instructor Verification Sheet for Lab L.2

Name: _____ **Date:** _____

1. Representation in a colormap of white and black.

   Instructor verification: _____

2. Vertical sinusoidal image.

   Instructor verification: _____

3. Horizontal higher frequency image. Give the frequency.

   Instructor verification: _____

4. Horizontal and vertical sinusoidal image.

   Instructor verification: _____

5. Compression ratio.

   Instructor verification: _____

6. Representation in truecolor of white and black.

   Instructor verification: _____

L.3 State machines

State machines are sequential. They begin in a starting state, and react to a sequence of inputs by sequentially transitioning between states. Implementation of state machines in software is fairly straightforward. In this lab, we explore doing this systematically, and build up to an implementation that composes two state machines.

L.3.1 Background

Strings in MATLAB

State machines operate on sequences of symbols from an alphabet. Sometimes, the alphabet is numeric, but more commonly, it is a set of arbitrary elements with names that suggest their meaning. For example, the input set for the answering machine in figure 3.1 is

$$Inputs = \{ring, offhook, end\ greeting, end\ message, absent\}.$$

Each element of the above set can be represented in MATLAB as a string (try `help strings`). Strings are surrounded by single quotes. For example,

```
>> x = 'ring';
```

The string itself is an array of characters, so you can index individual characters, as in

```
>> x(1:3)

ans =

rin
```

You can join strings just as you join ordinary arrays,

```
>> y = 'the';
>> z = 'bell';
>> [x, y, z]

ans =

ringthebell
```

However, this is not necessarily what you want. You may want instead to construct an array of strings, where each element of the array is a string (rather than a character). Such a collection of strings can be represented in MATLAB as a **cell array**,

```
>> c = {'ring' 'offhook' 'end greeting' 'end message' 'absent'};
```

Notice the curly braces instead of the usual square braces. A cell array in MATLAB is an array where the elements of the array are arbitrary MATLAB objects (such as strings and arrays). Cell arrays are indexed like ordinary arrays, so

```
>> c(1)

ans =

    'ring'
```

Often, you wish to test a string to see whether it is equal to some string. You usually cannot compare strings or cells of a cell array using `==`, as illustrated here:

```
>> c = 'ring';
>> if (c == 'offhook') result = 1; end
??? Error using ==> ==
Array dimensions must match for binary array op.

>> c = {'ring' 'offhook' 'end greeting' 'end message' 'absent'};
>> if (c(1) == 'ring') result = 1; end
??? Error using ==> ==
Function '==' not defined for variables of class 'cell'.
```

Strings should instead be compared using `strcmp` or `switch` (see the online help for these commands).

M-files

In MATLAB, you can save programs in a file and execute them from the command line. The file is called an **m-file**, and has a name of the form *command*.m, where *command* is the name of the command that you enter on the command line to execute the program.

You can use any text editor to create and edit m-files, but the one built into MATLAB is probably the most convenient. To invoke it, select "New" and "M-file" under the "File" menu.

To execute your program, MATLAB needs to know where to find your file. The simplest way to handle this is to make the current directory in MATLAB the same as the directory storing the m-file. For example, if you put your file in the directory

```
D:\users\eal
```

then the following will make the file visible to MATLAB:

```
>> cd D:\users\eal
>> pwd
```

```
ans =

D:\users\eal
```

The `cd` command instructs MATLAB to change the current working directory. The `pwd` command returns the current working directory (probably the mnemonic is *present* working directory).

You can instruct MATLAB to search through some sequence of directories for your m-files, so that they do not have to all be in the same directory. See `help path`. For example, instead of changing the current directory, you could type

```
path(path, 'D:\users\eal');
```

This command tells MATLAB to search for functions wherever it was searching before (the first argument `path`) and also in the new directory.

Suppose you create a file called `hello.m` containing

```
% HELLO - Say hello.
   disp('Hello');
```

The first line is a comment. The `disp` command displays its argument without displaying a variable name. On the command line, you can execute this

```
>> hello
Hello
```

Command names are not case sensitive, so `HELLO` is the same as `Hello` and `hello`. The comment in the file is used by MATLAB to provide online help. Thus,

```
>> help hello

  HELLO - Say hello.
```

The M-file above is a program, not a function. There is no returned value. To define a function, use the `function` command in your m-file. For example, store the following in a file `reverse.m`:

```
function result = reverse(argument)
% REVERSE - return the argument array reversed.
result = argument(length(argument):-1:1);
```

Then try:

```
>> reverse('hello')

ans =

olleh
```

The returned value is the string argument reversed.

A function can have any number of arguments and returned values. To define a function with two arguments, use the syntax

```
function [result1, result2] = myfunction(arg1, arg2)
```

and then assign values to `result1` and `result2` in the body of the file. To use such function on the command line, you must assign each of the return values to a variable as follows:

```
>> [r1, r2] = myfunction(a1, a2);
```

The names of the arguments and result variables are arbitrary.

L.3.2 *In-lab section*

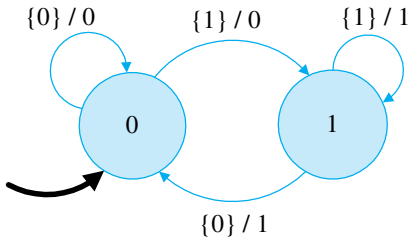
1. Write a for loop that counts the number of occurrences of `'a'` in

   ```
   >> d = {'a' 'b' 'a' 'a' 'b'};
   ```

   Then define a function `count` that counts the number of occurrences of `'a'` in any argument. How many occurrences are there in the following two examples?

   ```
   >> x = ['a', 'b', 'c', 'a', 'aa'];
   >> y = {'a', 'b', 'c', 'a', 'aa'};
   >> count(x)

   ans =

        ??

   >> count(y)

   ans =

        ??
   ```

   Why are they different?
2. The `input` function can be used to interactively query the user for input. Write a program that repeatedly asks the user for a string and then uses your `count` function to report the number of occurrences of `'a'` in the string. Write the program so that if the user enters `quit` or `exit`, the program exits, and otherwise, it asks for another input. Hint: Try `help while` and `help break`.
3. Consider the state machine in figure L.4. Construct an m-file containing a definition of its *update* function. Then construct an m-file containing a program that queries the user for an input, then if the input is in the input alphabet of the machine, uses it to react, and then asks the user for another



Inputs = {0, 1, *absent*}
Outputs = {0, 1, *absent*}

FIGURE L.4: A simple state machine.

input. If the input is not in the input alphabet, the program should assume the input is *absent* and stutter. Be sure that your update function handles stuttering.

L.3.3 *Independent section*

1. Design a virtual pet,* in this case a cat, by constructing a state machine, writing an *update* function, and writing a program to repeatedly execute the function, as in previous (3). The cat should behave as follows:

   > It starts out *happy*. If you *pet* it, it *purrs*. If you *feed* it, it *throws up*. If *time passes*, it gets *hungry* and *rubs* against your legs. If you feed it when it is hungry, it purrs and gets happy. If you pet it when it is hungry, it *bites* you. If time passes when it is hungry, it *dies*.

   The italicized phrases in this description should be elements in either the state space or the input or output alphabets. Give the input and output alphabets and a state transition diagram. Define the *update* function in MATLAB, and write a program to execute the state machine until the user types "`quit`" or "`exit`."
2. Construct a state machine that provides inputs to your virtual cat so that the cat never dies. In particular, your state machine should generate *time passes* and *feed* outputs in such a way that the cat never reaches the *dies* state.

   Note that this new state machine does not have particularly meaningful inputs. You can let the input alphabet be

$$Inputs = \{1, absent\}$$

* This problem is inspired by the Tamagotchi virtual pet made by Bandai in Japan. Tamagotchi, which translates as "cute little egg," were extremely popular in the late 1990s, and had behavior considerably more complex than that described in this exercise.

where an input of 1 indicates that the machine should output a nonstuttering output, and an input of *absent* means it should output a stuttering output.

Write a program where your feeder state machine is composed in cascade with your cat state machine, and verify (experimentally) that the cat does not die. Your state machine should allow time to pass (by producing an infinite number of *time passes* outputs) but should otherwise be as simple as possible.

Note that a major point of this exercise is to show that systematically constructed state machines can be very easily composed.

The feeder state machine is called an **open-loop controller** because it controls the pet without observing the output of the pet. For most practical systems, it is not possible to design an open-loop controller. The next lab explores closed-loop controllers.

Instructor Verification Sheet for Lab L.3

Name: _____ **Date:** _____

1. Count the number of occurrences of `a`. Understand the difference between a cell array and an array.

   Instructor verification: _____

2. Write a program with an infinite loop and user input.

   Instructor verification: _____

3. Construct and use *update* function.

   Instructor verification: _____

L.4 Control systems

This lab extends the previous one by introducing nondeterminism and feedback. In particular, you will modify the virtual pet that you constructed last time so that it behaves nondeterministically. The modification will make it impossible to keep the pet alive by driving it with another state machine in a cascade composition. You will instead have to use a feedback composition.

This scenario is typical of a control problem. The pet is a system to be controlled, with the objective of keeping it alive. You will construct a controller that observes the output of the virtual pet, and based on that output, constructs an appropriate input that will keep the pet alive. Since this controller observes the output of the pet, and provides input to the pet, it is called a **closed-loop controller**.

L.4.1 Background

Nondeterministic state machines have a *possibleUpdates* function rather than an *update* function. The *possibleUpdates* function returns a set of possible updates. You will construct this function to return a cell array, which was explored in the previous lab.

A software implementation of a nondeterministic state machine can randomly choose from among the results returned by *possibleUpdates*. It could conceptually flip coins to determine which result to choose each time. In software, the equivalent of coin flips is obtained through **pseudo-random number generators**. The MATLAB function `rand` is just such a pseudo-random number generator. The way that it works is that each time you use it, it gives you a new number (try `help rand`).

For this lab, you will need to be able to use cell arrays in more sophisticated ways than in the previous lab. Recall that a cell array is like an ordinary array, except that the elements of the array can be arbitrary MATLAB objects, including strings, arrays, or even cell arrays. A cell array can be constructed using curly braces instead of square brackets, as in

```
>> letters = {'a', 'b', 'c', 'd', 'e'};
>> whos letters
  Name          Size         Bytes  Class

  letters       1x5            470  cell array

Grand total is 10 elements using 470 bytes
```

The elements of the cell array can be accessed like elements of any other array, but there is one subtlety. If you access an element in the usual way, the result is a cell array, which might not be what you expect. For example,

```
>> x = letters(2)

x =

    'b'

>> whos x
  Name      Size         Bytes  Class

  x         1x1             94  cell array

Grand total is 2 elements using 94 bytes
```

To access the element as a string (or whatever the element happens to be), then use curly braces when indexing the array, as in

```
>> y = letters{2}

y =

b

>> whos y
  Name      Size         Bytes  Class

  y         1x1              2  char array

Grand total is 1 elements using 2 bytes
```

Notice that now the result is a character array rather than a 1×1 cell array.

You can also use curly braces to construct a cell array piece by piece. Here, for example, we construct and display a two-dimensional cell array of strings, and then access one of the elements as a string.

```
>> t{1,1} = 'upper left';
>> t{1,2} = 'upper right';
>> t{2,1} = 'lower left';
>> t{2,2} = 'lower right';
>> t

t =

    'upper left'    'upper right'
    'lower left'    'lower right'

>> t{2,1}
```

```
ans =

lower left
```

You can find out the size of a cell array in the usual way for arrays:

```
>> [rows, cols] = size(t)

rows =

     2


cols =

     2
```

You can also extract an entire row or column from the cell array the same way you do it for ordinary arrays, using ":" in place of the index. For example, to get the first row, do

```
>> t(1,:)

ans =

    'upper left'    'upper right'
```

L.4.2 *In-lab section*

1. Construct a MATLAB function `select` that, given a cell array with one row as an argument, returns a randomly chosen element of the cell array. Use your function to generate a random sequence of 10 letters from the cell array

   ```
   >> letters = {'a', 'b', 'c', 'd', 'e'};
   ```

   Hint: The MATLAB function `floor` combined with `rand` might prove useful to get random indexes into the cell array.
2. Construct a MATLAB function `chooserow` that, given a cell array with one or more rows, randomly chooses one of the rows and returns it as a cell array. Apply your function a few times to the `t` array that we constructed above.
3. A nondeterministic state machine has a *possibleUpdates* function rather than *updates*. This function returns a set of pairs, where each pair is a new state and an output.

   A convenient MATLAB implementation is a function that returns a two-dimensional cell array, with each of the possible updates on one row. As a first step toward this, modify your realization of the *update* function for the virtual cat of the previous lab so that it returns a 1×2 cell array with the

next state and output. Also modify your program that runs the cat (without the driver) so that it uses your new function. Verify that the cat still works properly.

4. Now modify the cat's behavior so that if it is hungry and you feed it, it sometimes gets happy and purrs (as it did before), but it sometimes stays hungry and rubs against your legs (i.e., change your *update* function so that if the state is hungry and you feed the cat, then return a 2×2 cell array where the two rows specify the two possible next state, output pairs). Modify the program that runs the cat to use your `chooserow` function to choose from among the options.
5. Compose your driver machine from the previous lab with your nondeterministic cat, and verify that the driver no longer keeps the cat alive. In fact, no open-loop controller will be able to keep the cat alive and allow time to pass. In the independent section of this lab, you will construct a **closed-loop controller** that keeps the cat alive. It is a feedback composition of state machines.

L.4.3 *Independent section*

Design a deterministic state machine that you can put into a feedback composition with your nondeterministic cat so that the cat is kept alive and time passes. Give the state transition diagram for your state machine and write a MATLAB function that implements its *update* function. Write a MATLAB program that implements the feedback composition.

Note that your program that implements the feedback composition faces a challenging problem. When the program starts, neither the inputs to the controller machine nor the inputs to the cat machine are available. So neither machine can react. For your controller machine, you should define MATLAB functions for both *update*, which requires a known input, and *output*, which does not. The *output* function, given the current state, returns the output that will be produced by the next reaction, if it is known, or *unknown* if it is not known. In the case of your controller, it should always be known, or the feedback composition will not be well formed.

Verify (by running your program) that the cat does not die.

Instructor Verification Sheet for Lab L.4

Name: _____ **Date:** _____

1. Generated random sequence of letters using `select`.

   Instructor verification: _____

2. Applied `chooserow` to the `t` array.

   Instructor verification: _____

3. The cat still works with the update function returning a cell array.

   Instructor verification: _____

4. The nondeterministic cat sometimes stays hungry when fed.

   Instructor verification: _____

5. The nondeterministic cat dies under open-loop control.

   Instructor verification: _____

L.5 Difference equations

The purpose of this lab is to construct difference equation models of systems and to study their properties. In particular, we experimentally examine **stability** by constructing stable, unstable, and marginally stable systems. We will also introduce elementary complexity measures. The principal new MATLAB skill required to develop these concepts is matrix operations.

L.5.1 *In-lab section*

1. MATLAB is particularly good at matrix arithmetic. In this problem, we explore matrix multiplication (see Basics: Matrix arithmetic on page 175 of the text).

   (a) Consider the 2×2 matrix

   $$M = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

   Without using MATLAB, give M^n, for $n = 0, 1, 2, 3$. Recall that by mathematical convention, for any square matrix M, $M^0 = I$, the identity matrix, so in this case,

   $$M^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

   Guess the general form of the matrix M^n. That is, give an expression for each of the elements of the matrix M^n.

   (b) Use MATLAB to compute M^{25}. Was your guess correct? Calculate a few more values using MATLAB until your guess is correct.

   (c) If your guess was correct, try to show it using induction. That is, first show that your guess for M^n is correct for some fixed n, like, for example, $n = 0$. Then assume your guess for M^n is correct for some fixed n, and show that it is correct for M^{n+1}.

2. A vector is a matrix where either the number of rows is one (in the case of a **row vector**) or the number of columns is one (in the case of a **column vector**). Let

   $$b = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

   be a column vector. We can equally well write this $b = [2, 3]^T$, where the superscript T indicates that the row vector $[2, 3]$ is transposed to make a column vector.

(a) Create a column vector in MATLAB equal to b above. Multiply it by M, given in the previous problem. Try forming both bM and Mb. Why does only one of these two work?
(b) Create a row vector by transposing b. (Try `help transpose` or look up "transpose" in the help desk.) Multiply this transpose by M. Try both $b^T M$ and Mb^T. Why does only one of them work?

3. Consider a two-dimensional difference equation system given by

$$A = \sigma \begin{bmatrix} \cos(\omega) & -\sin(\omega) \\ \sin(\omega) & \cos(\omega) \end{bmatrix}, \; b = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \; c = \sigma \begin{bmatrix} -\cos(\omega) \\ \sin(\omega) \end{bmatrix}, d = 0,$$

where $\omega, \sigma \in \mathit{Reals}$. Note that this is similar to the systems studied in exercises 10 and 14 of chapter 5, with the differences being the multiplying constant σ and the c vector. Let $\omega = \pi/8$ and plot the first 100 samples of the zero-state impulse response for the following values of σ.

(a) $\sigma = 1$.
(b) $\sigma = 0.95$.
(c) $\sigma = 1.05$.
(d) For which values of σ is the result periodic? What is the period? The system producing the periodic output is called an **oscillator**.
(e) You have constructed three distinct difference equation systems. One of these is a **stable system**, one is an **unstable system**, and one is a **marginally stable system**. Which is which? You can infer the answer from the ordinary English-language meaning of the word "stable." What will happen if the unstable system is allowed to continue to run beyond the 100 samples you calculated?

L.5.2 *Independent section*

1. In lab L.1, you constructed a sound waveform f: *Reals* $\rightarrow$ *Reals* given by

$$\forall\, t \in [0, 1], \quad f(t) = \exp(-5t)\sin(2\pi \times 440t).$$

You wrote a MATLAB script that calculated samples of this waveform at a sample rate of 8,000 samples/second. In this lab, we will construct the same waveform in a very different way, using difference equations.

Construct a difference equation system with impulse response given by

$$\forall\, n \in \mathit{Naturals}_0, \quad h(n) = \exp(-5n/8{,}000)\sin(2\pi \times 440n/8{,}000).$$

Give the matrix A, the vectors b, and c, and the scalar d of (5.33) and (5.34). Also give a MATLAB program that produces the first 8,000 samples of this

impulse response and plays it as a sound. Hint: You will need to understand what you did in problem 3 of the in-lab section.

2. For the system with the impulse response constructed in part 1, change the input so it consists of an impulse every 1/5 of a second (i.e., at an 8,000 samples/second sample rate),

$$x(n) = \begin{cases} 1 & \text{if } n \text{ is a multiple of 1,600} \\ 0 & \text{otherwise} \end{cases}$$

   Write a MATLAB script that plays two seconds of sound with this input. (Note: This is a simplified model of a guitar string being repeatedly plucked. The model is justifiable on physical grounds, although it is a fairly drastic simplification.)

3. Compare the complexity of the state machine model and the one you constructed in lab L.1. In particular, assuming in each case that you generate one second of sound at an 8,000 samples/second sample rate, count the number of scalar multiplications and additions that must be done to construct the sound vector. In the realization in lab L.1, you used the built-in MATLAB functions `exp` and `sin`. These functions are surprisingly expensive to compute, so count each evaluation of `exp` or `sin` on a scalar argument as 20 multiplications and 15 additions (they are actually typically more expensive even than this). You should find that the state machine realization is far less expensive by this measure. Do not count the cost of the MATLAB `sound` function, which we can't easily determine.

Instructor Verification Sheet for Lab L.5

Name: _____ **Date:** _____

1. Matrix multiplication in MATLAB, and induction demonstration.

   Instructor verification: _____

2. Matrix-vector multiplication.

   Instructor verification: _____

3. Sinusoids with exponential envelopes; stability.

   Instructor verification: _____

L.6 Differential equations

The purpose of this lab is to experiment with models of continuous-time systems that are described as differential equations. The exercises aim to solidify state-space concepts while giving some experience with software that models continuous-time systems.

The lab uses Simulink, a companion to MATLAB. The lab is self-contained, in the sense that no additional documentation for Simulink is needed. Instead, we rely on the online help facilities. Be warned, however, that these are not as good for Simulink as for MATLAB. The lab exercise will guide you, trying to steer clear of the more confusing parts of Simulink.

Simulink is a block-diagram modeling environment. As such, it has a more declarative flavor than MATLAB, which is imperative. 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.

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. There is quite a bit of sophistication in how this is done. The fact that you do not specify how it is done underscores the observation that Simulink has a declarative flavor.

The simulation is carried out by a **solver**, which examines the block diagram you have specified and constructs an execution that simulates its behavior. As you read the documentation and interact with the software, you will see various references to the solver. In fact, Simulink provides a variety of solvers, and many of these have parameters you can control. Indeed, simulation of continuous-time systems is generally inexact, and some solvers work better on some models than others. The models that we will construct work well with the default solver, so we need not be concerned with this (considerable) complication.

Simulink can also model discrete-time systems, and (a bit clumsily) mixed discrete and continuous-time systems. We will emphasize the continuous-time modeling because this cannot be done (conveniently) in MATLAB, and it is really the strong suit of Simulink.

L.6.1 Background

To run Simulink, start MATLAB and type `simulink` at the command prompt. This will open the Simulink library browser. To explore Simulink demos, at the MATLAB command prompt, type `demo`, and then find the Simulink item in the list that appears. To get an orientation about Simulink, open the help desk (using the Help menu), and find Simulink. Much of what is in the help desk will not be very useful to you. Find a section with a title "Building a Simple Model" or something similar and read that.

We will build models in state-space form, as in chapter 5, and as in the previous lab, but in continuous time. A continuous-time state-space model for a linear system has the form (see section 5.4):

$$\dot{z}(t) = Az(t) + bv(t) \tag{L.1}$$

$$w(t) = cz(t) + dv(t) \tag{L.2}$$

where

- z: *Reals* $\rightarrow$ *Reals*^{N} gives the state response;
- $\dot{z}(t)$ is the derivative of z evaluated at $t \in$ *Reals*;
- v: *Reals* $\rightarrow$ *Reals* is the input signal; and
- w: *Reals* $\rightarrow$ *Reals* is the output signal.

The input and output are scalars, so the models are SISO, but the state is a vector of dimension N, which in general can be larger than one. The derivative of a vector z is simply the vector consisting of the derivative of each element of the vector.

The principle that we will follow in modeling such a system is to use an Integrator block, which looks like this in Simulink:

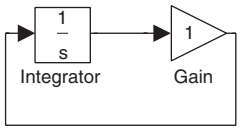
This block can be found in the library browser under "Simulink" and "Continuous." Create a new model by clicking on the blank-document icon at the upper left of the library browser, and drag an integrator into it. You should see the same icon as previously.

If the input to the integrator is $\dot{z}$, then the output is z (just think about what happens when you integrate a derivative). Thus, the pattern we will follow is to provide as the input to this block a signal $\dot{z}$.

We begin with a one-dimensional system ($N = 1$) in order to get familiar with Simulink. Consider the scalar differential equation

$$\dot{z}(t) = az(t) \tag{L.3}$$

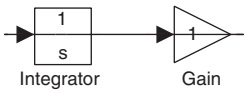
where $a \in$ *Reals* is a given scalar and z: *Reals* $\rightarrow$ *Reals* and $z(0)$ is some given initial state. We will set things up so that the input to the integrator is $\dot{z}$ and the output is z. To provide the input, however, we need the output, since $\dot{z}(t) = az(t)$. So we need to construct a feedback system that looks like this:



This model seems self-referential, and in fact it is, just as is (L.3).

Construct the preceding model. You can find the triangular "`Gain`" block in the library browser under "Simulink" and "Math." To connect the blocks, simply place the cursor on an output port and click and drag to an input port.

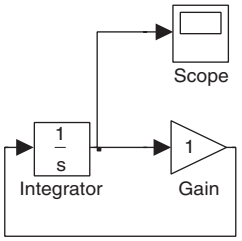
After constructing the feedback arc, you will likely see the following:



This is simply because Simulink is not very smart about routing your wires. You can stretch the feedback wire by clicking on it and dragging downward so that it does not go over top of the blocks.

This model, of course, has no inputs, no initial state, and no outputs, so it will not be very interesting to run it. You can set the initial state by double clicking on the integrator and filling in a value under "initial condition." Set the initial state to 1. Why is the initial state a property of the integrator? Because its output at time t is the state at time t. The "initial condition" parameter gives the output of the integrator when the model starts executing. Just like the feedback compositions of state machines in chapter 4, we need at least one block in the feedback loop whose output can be determined without knowing its input.

You will want to observe the output. To do this, find a block called "Scope" under "Simulink" and "Sinks" in the library browser, and drag it into your design. Connect it so that it displays the output of the integrator, as follows:



To make the connection, you need to hold the Control key while dragging from the output port of the integrator to the input port of the Scope. We are finished with the basic construction of the model. Now we can experiment with it.

L.6.2 In-lab section

1. Set the gain of the gain block by double-clicking on the triangular icon. Set it to -0.9. What value of a does this give you in the equation (L.3)?
2. Run the model for 10 time units (the default). To run the model, choose "Start" under the "Simulation" menu of the model window. To control the number of time units for the simulation, choose "Parameters" under the "Simulation" menu. To examine the result, double-click on the Scope icon. Clicking on the binoculars icon in the scope window will result in a better display of the result.
3. Write down analytically the function z given by this model. You can guess its form by examining the simulation result. Verify that it satisfies (L.3) by differentiating.
4. Change the gain block to have value 0.9 instead of -0.9 and re-run the model. What happens? Is the system stable? (Stable means that if the input is bounded for all time, then the output is bounded for all time. In this case, clearly the input is bounded since it is zero.) Give an analytical formula for z for this model.
5. Experiment with values of the gain parameter. Determine over what range of values the system is stable.

L.6.3 Independent section

Continuous-time linear state-space models are reasonable for some musical instruments. In this exercise, we will simulate an idealized and a more realistic tuning fork, which is a particularly simple instrument to model. The model will be a two-dimensional continuous-time state-space model.

Consider the state and output equations (L.1) and (L.2). Since the model is two dimensional, the state at each time is now a two-dimensional vector. The "initial condition" parameter of the Integrator block in Simulink can be given a vector. Set the initial value to the column vector

$$z(0) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}. \tag{L.4}$$

The factor A must be a 2×2 matrix if the state is a two-dimensional column vector. Unfortunately, the Gain block in Simulink cannot be given a matrix parameter. You must replace the Gain block with the MatrixGain block, also found in the "Math" library under "Simulink" in the library browser.

At first, we will assume there is no input, and we will examine the state response. Thus, we are only concerned at first with the simplified state equation

$$\dot{z}(t) = Az(t). \tag{L.5}$$

Recall that in chapter 2, equation (2.11) states that the displacement $x(t)$ at time t of a tine of the tuning fork satisfies the differential equation

$$\ddot{x}(t) = -\omega_0^2 x(t)$$

where ω_0 is constant that depends on the mass and stiffness of the tine, and where $\ddot{x}(t)$ denotes the second derivative with respect to time of x (see Probing Further: Physics of a Tuning Fork on page 61 of the text). This does not have the form of (L.5). However, we can put it in that form using a simple trick. Let

$$z(t) = \begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix}$$

and observe that

$$\dot{z}(t) = \begin{bmatrix} \dot{x}(t) \\ \ddot{x}(t) \end{bmatrix}.$$

Thus, we can write (L.5) as

$$\dot{z}(t) = \begin{bmatrix} \dot{x}(t) \\ \ddot{x}(t) \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix}$$

for suitably chosen constants $a_{1,1}$, $a_{1,2}$, $a_{2,1}$, and $a_{2,2}$.

1. Find $a_{1,1}$, $a_{1,2}$, $a_{2,1}$, and $a_{2,2}$ for the tuning fork model.
2. Use Simulink to plot the state response of the tuning fork when the initial state is given by (L.4). You will have to pick a value of ω_0. Use Simulink to help you find a value of ω_0 so that the state completes one cycle in 10 time units. Each sample of the state response has two elements. These represent the displacement and speed, respectively, of the tuning fork tine in the model. The displacement is what directly translates into sound.
3. Change ω_0 so that the state has a frequency of 440 Hz, assuming the time units are seconds. Change the simulation parameters so that you run the model through five complete cycles.
4. Change the simulation parameters so that you run the model through one second. Use the Simulink To Workspace block to write the result to the workspace, and then use the MATLAB `soundsc` function to listen to it. Note: You will need to set the sample time parameter of the `To Workspace` block to 1/8,000. You will also need to specify that the save format should be a matrix. For your lab report, print your block diagram and annotate it with all the parameters that have values different from the defaults.
5. In practice, a tuning fork will not oscillate forever as the model does. We can add damping by modifying the matrix A. Try replacing the zero value of $a_{2,2}$ with -10. What happens to the sound? This is called **damping**. Experiment

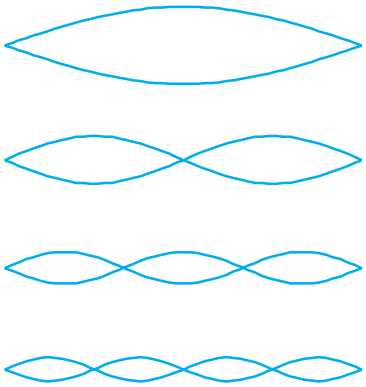


FIGURE L.5: Four modes of vibration of a guitar string.

with different values for $a_{2,2}$. Describe how the different values affect the sound. Determine (experimentally) for what values of $a_{2,2}$ the system is stable.

6. A tuning fork is not much of a musical instrument. Its sound is too pure (spectrally). A guitar string, however, operates on similar principles as the tuning fork, but has a much more appealing sound.

   A tuning fork vibrates with only one mode. A guitar string, however, vibrates with multiple modes, as illustrated in figure L.5. Each of these vibrations produces a different frequency. The top one in the figure produces the lowest frequency, called the **fundamental**, which is typically the frequency of the note being played, such as 440 Hz for A-440. The next mode produces a component of the sound at twice that frequency, 880 Hz; this component is called the **first harmonic**. The third produces three times the frequency, 1,320 Hz, and the fourth produces four times the fundamental, 1,760 Hz; these components are the second and third harmonics.

   If the guitar string is undamped, and the fundamental frequency is f_0 Hz, then the combined sound is a linear combination of the fundamental and the three (or more) harmonics. This can be written as a continuous-time function y where for all $t \in Reals$,

$$y(t) = \sum_{k=0}^{N} c_k sin(2\pi f_k t)$$

   where N is the number of harmonics and c_k gives the relative weights of these harmonics. The values of c_k will depend on the guitar construction and how it is played, and affect the **timbre** of the sound.

   The model you have constructed above generates a damped sinusoid at 440 Hz. Create a Simulink model that produces a fundamental of 440 Hz plus

three harmonics. Experiment with the amplitudes of the harmonics relative to the fundamental, as well as with the rates of decay of the four components. Note how the quality of the sound changes. Your report should include a printout of your model with the parameter values that you have chosen to get a sound like that of a plucked string.

Instructor Verification Sheet for Lab L.6

Name: _____ **Date:** _____

1. Value of *a.*

   Instructor verification: _____

2. Plot of the state response.

   Instructor verification: _____

3. Formula for function *z.* Verified by differentiating.

   Instructor verification: _____

4. Formula for function *z.*

   Instructor verification: _____

5. Range of values for the gain over which the system is stable.

   Instructor verification: _____

L.7 Spectrum

The purpose of this lab is to learn to examine the frequency domain content of signals. Two methods will be used. The first method will be to plot the discrete Fourier series coefficients of finite signals. The second will be to plot the Fourier series coefficients of finite segments of time-varying signals, creating what is known as a **spectrogram**.

L.7.1 Background

A finite discrete-time signal with p samples has a discrete-time Fourier series expansion

$$x(n) = A_0 + \sum_{k=1}^{(p-1)/2} A_k \cos(k\omega_0 n + \phi_k) \tag{L.6}$$

for p odd and

$$x(n) = A_0 + \sum_{k=1}^{p/2} A_k \cos(k\omega_0 n + \phi_k) \tag{L.7}$$

for p even, where $\omega_0 = 2\pi/p$.

A finite signal can be considered to be one cycle of a periodic signal with fundamental frequency ω_0, in units of radians/sample, or $1/p$ in Hertz. In this lab, we will assume p is always even, and we will plot the magnitude of each of the frequency components, $|A_0|, \cdots, |A_{p/2}|$ for each of several signals, in order to gain intuition about the meaning of these coefficients.

Notice that each $|A_k|$ gives the amplitude of the sinusoidal component of the signal at frequency $k\omega_0 = k2\pi/p$, which has units of radians/sample. In order to interpret these coefficients, you will probably want to convert these units to Hertz. If the sampling frequency is f_s samples/second, then you can do the conversion as follows (see box on page 248 of the text):

$$\frac{(k2\pi/p)\,[\text{radians/sample}]\, f_s\,[\text{samples/second}]}{2\pi\,[\text{radians/cycle}]} = kf_s/p\,[\text{cycles/second}]$$

Thus, each $|A_k|$ gives the amplitude of the sinusoidal component of the signal at frequency kf_s/p Hz.

Note that MATLAB does not have any built-in function that directly computes the discrete Fourier series coefficients. However, it does have a realization of the fast Fourier transform, a function called `fft`, which can be used to construct

the Fourier series coefficients. In particular, `fourierSeries` is a function that returns the DFS coefficients:*

```
function [magnitude, phase] = fourierSeries(x)
% FOURIERSERIES - Return the magnitude and phase of each
% sinusoidal component in the Fourier series expansion for
% the argument, which is interpreted as one cycle of a
% periodic signal.  The argument is assumed to have an
% even number p of samples. The first returned value is an
% array containing the amplitudes of the sinusoidal
% components in the Fourier series expansion, with
% frequencies 0, 1/p, 2/p, ... 1/2. The second returned
% value is an array of phases for the sinusoidal
% components. Both returned values are arrays with length
% (p/2)+1.
p = length(x);
f = fft(x)/p;
magnitude(1) = abs(f(1));
upper = p/2;
magnitude(2:upper) = 2*abs(f(2:upper));
magnitude(upper+1) = abs(f(upper+1));
phase(1) = angle(f(1));
phase(2:upper) = angle(f(2:upper));
phase(upper+1) = angle(f(upper+1));
```

In particular, if you have an array `x` with even length,

```
[A, phi] = fourierSeries(x);
```

returns the DFS coefficients in a pair of vectors.

To plot the magnitudes of the Fourier series coefficients versus frequency, you can simply say

```
p = length(x);
frequencies = [0:fs/p:fs/2];
plot(frequencies, A);
xlabel('frequency in Hertz');
ylabel('amplitude');
```

where `fs` has been set to the sampling frequency (in samples per second). The line

```
frequencies = [0:fs/p:fs/2];
```

* This function can be found at http://www.aw.com/lee_varaiya/matlab/fourierSeries.m.

bears further examination. It produces a vector with the same length as `A`, namely $1 + p/2$, where p is the length of the `x` vector. The elements of the vector are the frequencies in Hertz of each Fourier series component.

L.7.2 *In-lab section*

1. To get started, consider the signal generated by

```
t = [0:1/8000:1-1/8000];
x = sin(2*pi*800*t);
```

This is 8,000 samples of an 800-Hz sinusoid sampled at 8,000 samples/second. Listen to it. Use the `fourierSeries` function as described above to plot the magnitude of its discrete Fourier series coefficients. Explain the plot.

Consider the continuous-time sinusoid

$$x(t) = \sin(2\pi 800t).$$

The preceding `x` vector is 8,000 samples of this sinusoid taken at a sample rate of 8 kHz. Notice that the frequency of the sinusoid is the derivative of the argument to the sine function,

$$\omega = \frac{d}{dt} 2\pi 800t = 2\pi 800$$

in units of radians per second. This fact will be useful when looking at more interesting signals.

2. With the same `t`, consider the more interesting waveform generated by

```
y = sin(2*pi*800*(t.*t));
```

This is called a **chirp**. Listen to it. Plot its Fourier series coefficients using the `fourierSeries` function as described.

This chirp is 8-kHz samples of the continuous-time waveform

$$y(t) = \sin(2\pi 800t^2).$$

The **instantaneous frequency** of this waveform is defined to be the derivative of the argument to the sine function,

$$\omega(t) = \frac{d}{dt} 2\pi 800t^2 = 4\pi 800t.$$

For the given values `t` used to compute samples `y`, what is the range of instantaneous frequencies? Explain how this corresponds with the plot of the Fourier series coefficients, and how it corresponds with what you hear.

3. The Fourier series coefficients computed in part 2 describe the range of instantaneous frequencies of the chirp pretty well, but they do not describe the dynamics very well. Plot the Fourier series coefficients for the waveform given by

```
z = y(8000:-1:1);
```

   Listen to this sound. Does it sound the same as y? Does its Fourier series plot look the same? Why?

4. The chirp signal has a dynamically varying frequency-domain structure. More precisely, there are certain properties of the signal that change slowly enough that our ears detect them as a change in the frequency structure of the signal rather than as part of the frequency structure (the timbre or tonal content). Recall that our ears do not hear sounds below about 30 Hz. Instead, the human brain hears changes below 30 Hz as variations in the nature of the sound rather than as frequency domain content. The preceding Fourier series methods fail to reflect this psychoacoustic phenomenon.

   A simple fix is the **short-time Fourier series**. The preceding chirp signals have 8,000 samples, lasting one second. But since we don't hear variations below 30 Hz as frequency content, it probably makes sense to reanalyze the chirp signal for frequency content 30 times in the one second. This can be done using the following function:*

```
function waterfallSpectrogram(s, fs, sizeofspectra, numofspectra)

% WATERFALLSPECTROGRAM - Display a 3-D plot of a spectrogram
% of the signal s.
%
% Arguments:
%   s - The signal.
%   fs - The sampling frequency (in samples per second).
%   sizeofspectra - The number of samples to use to calculate each
%          spectrum.
%   numofspectra - The number of spectra to calculate.

frequencies = [0:fs/sizeofspectra:fs/2];
offset = floor((length(s)-sizeofspectra)/numofspectra);
for i=0:(numofspectra-1)
    start = i*offset;
    [A, phi] = fourierSeries(s((1+start):(start+sizeofspectra)));
    magnitude(:,(i+1)) = A';
end
waterfall(frequencies, 0:(numofspectra-1), magnitude');
```

* This code can be found at http://www.aw.com/lee_varaiya/matlab/waterfallSpectrogram.m

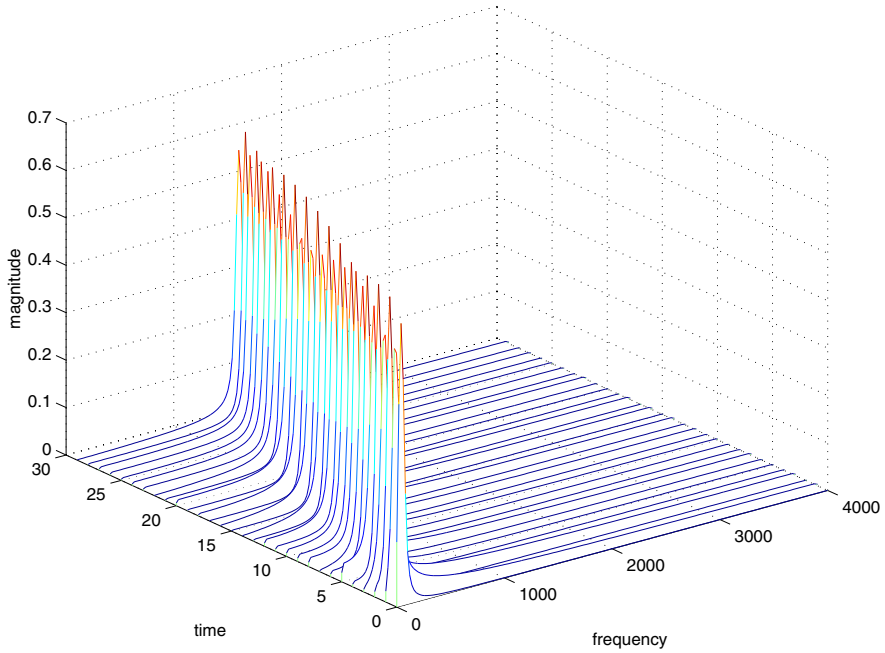


FIGURE L.6: Time varying discrete Fourier series analysis of a chirp.

```
xlabel('frequency');
ylabel('time');
zlabel('magnitude');
```

To invoke this function on the chirp, do

```
t = [0:1/8000:1-1/8000];
y = sin(2*pi*800*(t.*t));
waterfallSpectrogram(y, 8000, 400, 30);
```

which yields the plot shown in figure L.6. That plot shows 30 distinct sets of Fourier series coefficients, each calculated using 400 of the 8,000 available samples. Explain how this plot describes the sound you hear. Create a similar plot for the reverse chirp, signal `z` given in part 3.

5. Figure L.6 is reasonably easy to interpret because of the relatively simple structure of the chirp signal. More interesting signals, however, become very hard to view this way. An alternative visualization of the frequency content of such signals is the **spectrogram**. A spectrogram is a plot like that in figure L.6, but looking straight down from above the plot. The height of each point is depicted by a color (or intensity, in a gray-scale image) rather than by height. You can generate a spectrogram of the chirp as follows:

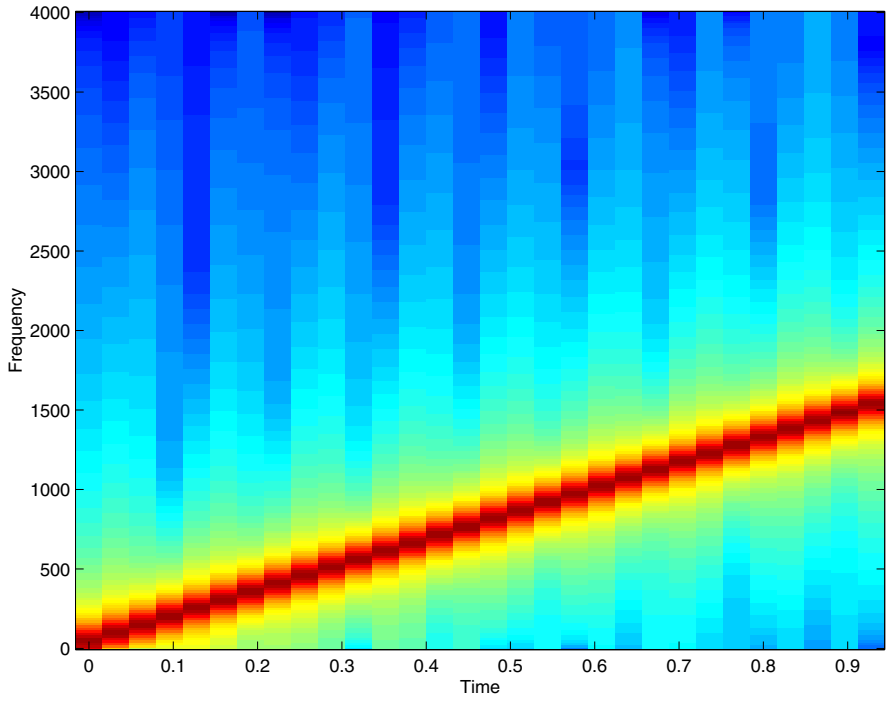


FIGURE L.7: Spectrogram of the chirp signal.

```
specgram(y,512,8000);
```

With the additionala command `colormap(gray)256))`, the result is the image shown in figure L.7. You could experiment with different colormaps for rendering this spectrogram by using the `colormap` command. A particularly useful one is `hot`, obtained by the command

```
colormap(hot);
```

Create a similar image for the reverse chirp, `z`, of part 3.

6. A number of audio files are available at

```
http://www.aw.com/lee_varaiya/sounds
```

In Netscape, you can save these to your local computer disk by placing the mouse on the file name, clicking with the right mouse button, and selecting "Save Link As." For example, if you save `voice.au` to your current working directory, then in MATLAB you can do

```
y = auread('voice.au');
soundsc(y)
subplot(2,1,1); specgram(y,1024,8000,[],900)
```

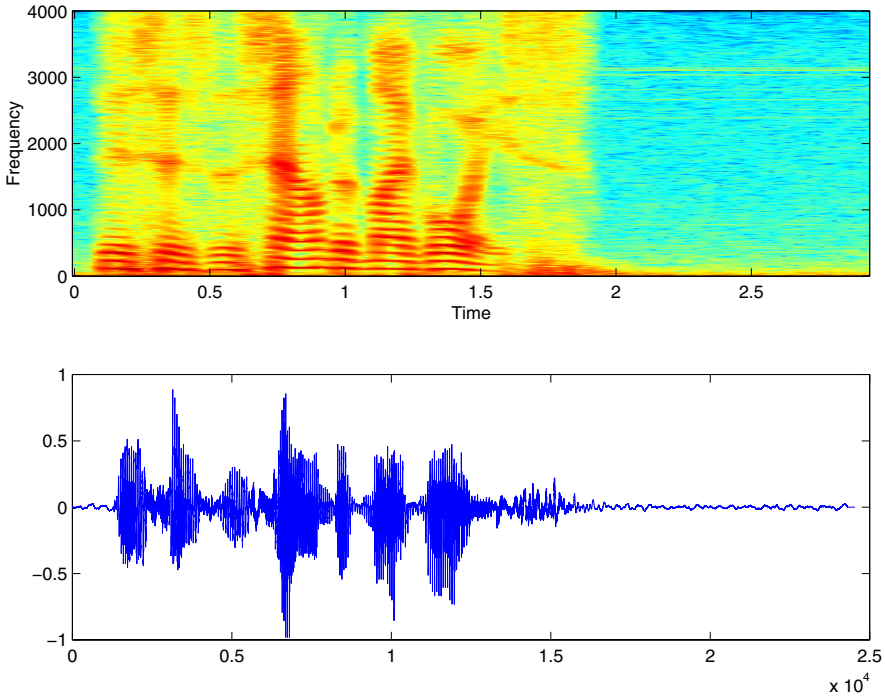


FIGURE L.8: Spectrogram and plot of a voice segment (one of the authors saying "this is the sound of my voice").

```
colormap(gray(256))
subplot(2,1,2); plot(y)
```

to get the result shown in figure L.8. Use this technique to get similar results for other sound files in the same directory. Interpret the results.

L.7.3 *Independent section*

1. For the chirp signal as given in the preceding discussion,

   ```
   y = sin(2*pi*800*(t.*t));
   ```

   generate the discrete Fourier series coefficients using `fourierSeries` as explained in section L.7.1. Then, write a MATLAB function that uses (L.7) to reconstruct the original signal from the coefficients. Your MATLAB function should begin as follows:

   ```
   function x = reconstruct(magnitude, phase)
   % RECONSTRUCT - Given a vector of magnitudes and a vector
   % of phases, construct a signal that has these magnitudes
   ```

```
% and phases as its discrete Fourier series coefficients.
% The arguments are assumed to have odd length, p/2 + 1,
% and the returned vector will have length p.
```

Note that this function will require a large number of computations. If your computer is not up to the task, the construct the Fourier series coefficients for the first 1,000 samples instead of all 8,000, and reconstruct the original from those coefficients. To check that the reconstruction works, subtract your reconstructed signal from y and examine the difference. The difference will not be perfectly zero, but it should be very small compared to the original signal. Plot the difference signal.

2. In the remainder of this lab, we will study **beat signals**, which are combinations of sinusoidal signals with closely spaced frequencies. First, we need to develop some background.

   Use Euler's relation to show that

   $$2\cos(\omega_c t)\cos(\omega_\Delta t) = \cos((\omega_c + \omega_\Delta)t) + \cos((\omega_c - \omega_\Delta)t).$$

   for any ω_c, ω_Δ, and t in *Reals*. Hint: See Basics: Sinusoids on page 294 of the text.

   A consequence of this identity is that if two sinusoidal signals with different frequencies, ω_c and ω_Δ, are multiplied together, the result is the same as if two sinusoids with two other frequencies, $\omega_c + \omega_\Delta$ and $\omega_c - \omega_\Delta$, are added together.
3. Construct the sum of two cosine waves with frequencies of 790 and 810 Hz. Assume the sample rate is 8 kHz, and construct a vector in MATLAB with 8,000 samples. Listen to it. Describe what you hear. Plot the first 800 samples (1/10 second). Explain how the plot illustrates what you hear. Explain how the identity in part 2 explains the plot.
4. What is the period of the waveform in part 3? What is the fundamental frequency for its Fourier series expansion? Plot its discrete Fourier series coefficients (the magnitude only) using `fourierSeries`. Plot its spectrogram using `specgram`. Choose the parameters of `specgram` so that the warble is clearly visible. Which of these two plots best reflects perception?

Instructor Verification Sheet for Lab L.7

Name: _____ **Date:** _____

1. Plot of the DFS coefficients of the sinusoid, with explanation.

   Instructor verification: _____

2. Plot of the DFS, plus range of instantaneous frequencies, plus correspondence with the sound.

   Instructor verification: _____

3. Plot of the DFS is the same, yet the sound is different. Explanation.

   Instructor verification: _____

4. Explain how figure L.6 describes the sound you hear. Plot the reverse chirp.

   Instructor verification: _____

5. Create and interpret a spectrogram for at least one other sound file.

   Instructor verification: _____

L.8 Comb filters

The purpose of this lab is to use a kind of filter called a comb filter to deeply explore concepts of impulse response and frequency response.

The lab uses Simulink, like lab L.6. Unlike lab L.6, it will use Simulink for discrete-time processing. Be warned that of this writing discrete-time processing is not the best part of Simulink, so some operations will be awkward. Moreover, the blocks in the block libraries that support discrete-time processing are not well organized. It can be difficult to discover how to do something as simple as an N-sample delay or an impulse source. We will identify the blocks you will need.

The lab is self-contained in the sense that no additional documentation for Simulink is needed. As in lab L.6, be warned that the online documentation is not as good for Simulink as for MATLAB. You will want to follow our instructions closely, or you are likely to discover very puzzling behavior.

L.8.1 Background

To run Simulink, start MATLAB and type `simulink` at the command prompt. This will open the Simulink library browser. The library browser is a hierarchical listing of libraries with blocks. The names of the libraries are (usually) suggestive of the contents, although sometimes blocks are found in surprising places, and some of the libraries may have meaningless names (such as "Simulink").

Here, we explain some of the techniques you will need to implement the lab. You may wish to skim these now and return them when you need them.

Simulation parameters

First, since we will be processing audio signals with a sample rate of 8,000 samples per second, you need to force Simulink to execute the model as a discrete-time model with sample rate 8,000 samples per second (recall that Simulink excels at continuous-time modeling). Open a blank model by clicking on the document icon at the upper left of the library browser window. Find the Simulation menu in that window, and select Parameters. Set the parameters so that the window looks like what is shown in figure L.9. Specifically, set the stop time to 4.0 (seconds), the solver options to "Fixed-step" and "discrete (no continuous states)," and the fixed step size to 1/8,000.

Reading and writing audio signals

Surprisingly, Simulink is more limited and awkward than MATLAB in its ability to read and write audio files. Consequently, the following will seem like more trouble than it is worth. Bear with us. As of this writing, Simulink only supports Microsoft wave files, which typically have the suffix ".wav". You may obtain a suitable audio file for this lab at

```
http://www.aw.com/lee_varaiya/sounds/voice.wav
```

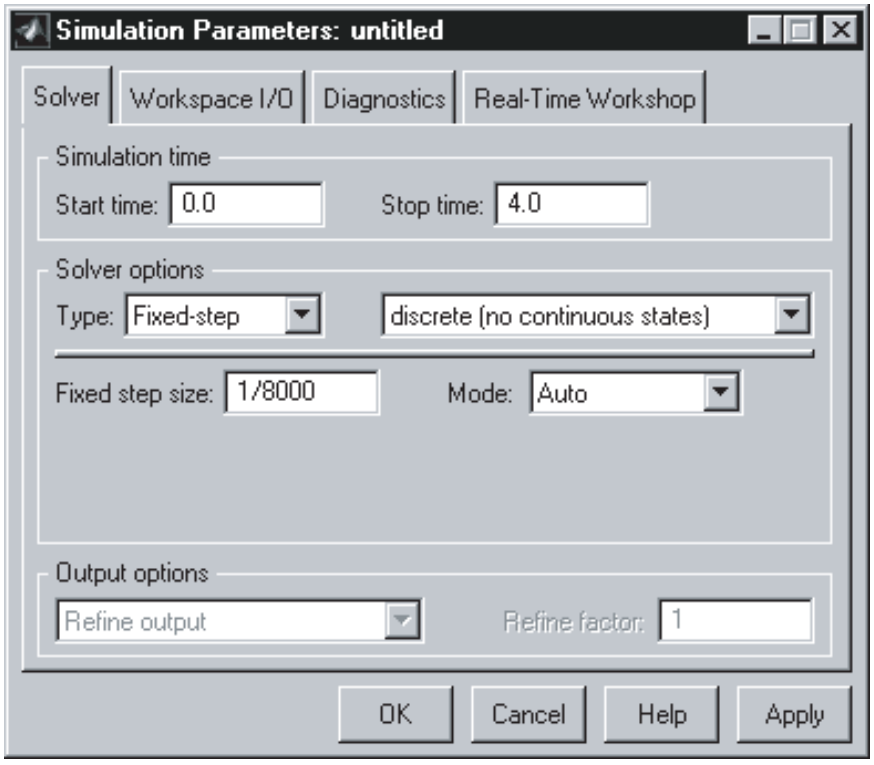


FIGURE L.9: Simulation parameters for discrete-time audio processing in Simulink.

In Netscape you can go to

```
http://www.aw.com/lee_varaiya/sounds/
```

and then right-click on the `voice.wav` file name to bring up a menu, and choose "Save Link As..." to save the file to your local disk. It is best in the MATLAB command window to then change the current working directory to the one in which you stored the file using the `cd` command. This will make it easier to use the file.

To make sure we can process audio signals, create the test model shown in figure L.10. To do this, in a new model window with the simulation parameters set as explained in "Simulation Parameters" on page 52, create an instance of the block called `From Wave File`. This block can be found in the library browser under `DSP Blockset` and `DSP Sources`. Set the parameters of that block to

```
File name: voice.wav
Samples per frame: 1
```

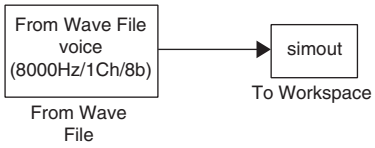


FIGURE L.10: Test model for Simulink audio.

The first parameter assumes you have set the current working directory to the directory containing the `voice.wav` file. The second indicates to Simulink that it should produce audio samples one at a time, rather than collecting them into vectors to produce many at once.

Next, find the `To Workspace` block in the Simulink block library, under Sinks. Create an instance of that block in your model. Edit its parameters to change the "Save format" to "Array" (or "Matrix," depending on your MATLAB version). You can leave other parameters at their default values.

Connect the blocks as shown in figure L.10.

Assuming the simulation parameters have been set as explained in "Simulation Parameters" on page 52, you can now run the model by invoking the Start command under the Simulation menu. This will result in a new variable called `simout` appearing in the MATLAB workspace. In the MATLAB command window, do

```
soundsc(simout)
```

to listen to the voice signal.

Note that the `DSP Sinks` library has a block called `To Wave Device`, which in theory will produce audio directly to the audio device. In practice, however, it seems much easier to use the `To Workspace` block and the `soundsc` command. For one thing, `soundsc` scales the audio signal automatically. It also circumvents difficulties with real-time performance, platform dependence problems, and ideosyncrasies with buffering. However, if you wish to try the `To Wave Device` block, and can figure out how to get it to work, feel free to use it.

L.8.2 *In-lab section*

1. Consider the equation

$$\forall\, n \in \textit{Integers}, \quad y(n) = x(n) + \alpha y(n - N) \tag{L.8}$$

for some real constant $\alpha < 1$ and integer constant $N > 0$. Assume the sample rate is 8,000 samples per second. The input is $x(n)$ and the output is $y(n)$. The equation describes an LTI system where the output is delayed, scaled, and fed back. Such a system is called a **comb filter**, for reasons that will become apparent in this lab. The filter can be viewed as a feedback structure, as

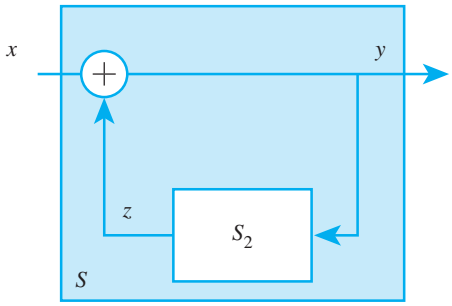


FIGURE L.11: Comb filter modeled as a feedback system.

shown in figure L.11, where S_2 is a system with input y and output z. Give a similar equation describing S_2, relating y and z.

2. Implement in Simulink the comb filter from part (a). Provide as input the file `voice.wav` (see page 52). Send the output to the workspace, just like figure L.10, so that you can use `soundsc` to listen to the result. You will probably need the `Gain` and `Sum` blocks, which you can find in the Simulink, Math library. The delay in the feedback path can be implemented by the `Integer Delay` block, which you can find in the DSP Blockset, General DSP, Signal Operations library.

   Experiment with the values of N. Try $N = 2{,}000$ and $N = 50$ and describe qualitatively the difference. With $N = 50$, the effect is called a sewer pipe effect. Why? Can you relate the physics of sound in a sewer pipe with our mathematical model? Hint: The speed of sound in air is approximately

$$331.5 + 0.6T \text{meters/second}$$

   where T is the temperature in degress celcius. Thus, at 20 degrees, sound travels at about 343.7 meters/second. A delay of $N = 50$ samples at an 8,000 samples/second sample rate is equal to the time it takes sound to travel roughly 2 meters, twice the diameter of a 1-meter sewer pipe.

   Experiment with the value of α. What happens when $\alpha = 0$? What happens when $\alpha = 1$? When $\alpha > 1$? You may wish to plot the output in addition to listening to it.

3. Modify your Simulink model so that its output is the first second (the first 8,001 samples) of the impulse response of the system defined by (L.8), with $\alpha = 0.99$ and $N = 40$.

   The simplest approach is to provide an impulse as an input. To do that, use the `Discrete Pulse Generator` block, found in the Simulink, Sources. This block can be (sort of) configured to generate a Kronecker delta function. Set its amplitude to 1, its period to something longer than the total

number of samples (i.e., larger than 8,001), its pulse width to 1, its phase delay to 0, and its sample time to 1/8,000.

You will also want to change the simulation parameters to execute your system for 1 second instead of 4.

Listen to the impulse response. Plot it. Can you identify the tone that you hear? Is it a musical note? Hint: Over short intervals, a small fraction of a second, the impulse response is roughly periodic. What is its period?

4. In the next lab, you will modify the comb filter to generate excellent musical sounds resembling plucked strings, such as guitars. As a first step toward that goal, we can make a much less mechanical sound than the impulse response by initializing the delay with random data. Modify your Simulink model so that the comb filter has no input, and instead of an input, the `Integer Delay` block is given random initial conditions. Use $\alpha = 0.99$ and $N = 40$, and change the parameters of the `Integer Delay` block so that its initial conditions are given by

```
randn(1,40)
```

The MATLAB `randn` function returns a vector of random numbers (try `help randn` in the MATLAB command window).

Listen to the result. Compare it to the sound of the impulse response. It should be richer, and less mechanical, but should have the same tone. It is also louder (even though `soundsc` scales the sound).

L.8.3 *Independent section*

The comb filter is an LTI system. Figure L.11 is a special case of the feedback system considered in section 8.5.2, which is shown there to be LTI. Thus, if the input is

$$x(n) = e^{j\omega n}$$

then the output is

$$y(n) = H(\omega)e^{j\omega n}$$

where H: *Reals* $\rightarrow$ *Complex* is the frequency response. Find the frequency response of the comb filter. Plot the magnitude of the frequency response over the range 0 to 4 kHz using MATLAB. Why is it called a comb filter? Explain the connection between the tone that you hear and the frequency response.

Instructor Verification Sheet for Lab L.8

Name: _____ **Date:** _____

1. Found an equation for S_2, relating y and z.

   Instructor verification: _____

2. Constructed Simulink model and obtained both sewer pipe effect and echo effect.

   Instructor verification: _____

3. Constructed the impulse response and identified the tone.

   Instructor verification: _____

4. Created sound with random values in the feedback delay.

   Instructor verification: _____

L.9 Plucked string instrument

The purpose of this lab is to experiment with models of a plucked string instrument, using it to deeply explore concepts of impulse response, frequency response, and spectrograms. The methods discussed in this lab were invented by Karplus and Strong, and first reported in

> K. Karplus and A. Strong, "Digital Sythesis of Plucked-String and Drum Timbres," *Computer Music Journal*, vol. 7, no. 2, pp. 43-55, Summer 1983.

The lab uses Simulink, like lab L.8. It assumes you have understood that lab and the techniques it uses in detail.

L.9.1 Background

In the previous lab, you constructed in Simulink the feedback system shown in figure L.11, where S_2 was an N sample delay. In this lab, you will enhance the model by replacing S_2 with slightly more complicated filters. These filters will consist of the same N sample delay in cascade with two other filters, a **lowpass filter** and an **allpass filter**. The objective will be to get truly high-quality plucked string sounds.

Delays

Recall from example 8.9 that the N sample delay system has frequency response

$$H(\omega) = e^{-i\omega N}.$$

This same frequency response was obtained in example 9.10 by calculating the DTFT of the impulse response. Note that the magnitude response is particularly simple,

$$|H(\omega)| = 1.$$

Recall that this is an **allpass filter**.

The phase response is

$$\angle H(\omega) = -\omega N.$$

The phase response is a linear function of frequency, ω, with slope $-N$. A filter with such a phase response is said to have **linear phase**. A delay is particularly simple form of a linear phase filter. Notice that the amount of delay is the negative of the derivative of the phase response,

$$\frac{d\angle H(\omega)}{d\omega} = -N.$$

This fact will be useful when we consider more complicated filters than this simple delay.

Allpass filters

We will need a slightly more complicated allpass filter than the N sample delay. Consider a filter given by the following difference equation,

$$\forall\, n \in \textit{Integers}, \quad y(n) + ay(n-1) = ax(n) + x(n-1) \tag{L.9}$$

for some constant $0 < a \leq 1$. This defines an LTI system, so if the input is $x(n) = e^{i\omega n}$, then the output is $H(\omega)e^{i\omega n}$, where H is the frequency response. We can determine the frequency response using this fact by plugging this input and output into (L.9),

$$H(\omega)e^{i\omega n} + aH(\omega)e^{i\omega(n-1)} = ae^{i\omega n} + e^{i\omega(n-1)}.$$

This can be rewritten as

$$H(\omega)e^{i\omega n}(1 + ae^{-i\omega}) = e^{i\omega n}(a + e^{-i\omega}).$$

Eliminating $e^{i\omega n}$ on both sides we get

$$H(\omega)(1 + ae^{-i\omega}) = a + e^{-i\omega}.$$

Solving for $H(\omega)$ we get

$$H(\omega) = \frac{a + e^{-i\omega}}{1 + ae^{-i\omega}}. \tag{L.10}$$

We could immediately proceed to plotting the magnitude and phase response using MATLAB, but instead, we will first manipulate this further to get some insight. Being slightly tricky, we will multiply top and bottom by $e^{i\omega/2}$ to get

$$H(\omega) = \frac{ae^{i\omega/2} + e^{-i\omega/2}}{e^{i\omega/2} + ae^{-i\omega/2}}.$$

Now notice that the top and bottom are complex conjugates of one another. For example, let

$$b(\omega) = ae^{i\omega/2} + e^{-i\omega/2} \tag{L.11}$$

and note that

$$H(\omega) = \frac{b(\omega)}{b^*(\omega)}.$$

Since the numerator and denominator have the same magnitude,*

$$|H(\omega)| = 1.$$

The filter is allpass!

The phase response, however, is more complicated. Note that

$$\angle H(\omega) = \angle b(\omega) - \angle b^*(\omega).$$

But since for any complex number z, $\angle(z^*) = -\angle(z)$,

$$\angle H(\omega) = 2\angle b(\omega).$$

Thus, to find the phase response, we simply need to find $\angle b(\omega)$. Plugging Euler's relation into (L.11) we get

$$b(\omega) = (a+1)\cos(\omega/2) + i(a-1)\sin(\omega/2).$$

Since the angle of a complex number z is $\tan^{-1}(Im\{z\}/Re\{z\})$,

$$\angle H(\omega) = 2\tan^{-1}\left(\frac{(a-1)\sin(\omega/2)}{(a+1)\cos(\omega/2)}\right).$$

Since $\tan(w) = \sin(w)/\cos(w)$,

$$\angle H(\omega) = 2\tan^{-1}\left(\frac{a-1}{a+1}\tan(\omega/2)\right).$$

This form yields insight for small ω. In particular, when ω is small (compared to π),

$$\tan(\omega/2) \approx \omega/2,$$

so

$$\angle H(\omega) \approx 2\tan^{-1}\left(\frac{a-1}{a+1}\omega/2\right).$$

Since $0 < a \leq 1$, the argument to the arctangent is small if ω is small, so for low frequencies,

$$\angle H(\omega) \approx \frac{a-1}{a+1}\omega = -d\omega.$$

* For any two complex numbers z and w, note that $|z/w| = |z|/|w|$ and $\angle(z/w) = \angle(z) - \angle(w)$.

where d is defined by

$$d = -\frac{a-1}{a+1}. \tag{L.12}$$

Thus, at low frequencies, this allpass filter has linear phase with slope $-d$. At low frequencies, therefore, it is an allpass with linear phase, which means that it behaves exactly like a delay! However, unlike the N sample delay, the amount of delay is d, which depending on a can be any real number between 0 and 1. Thus, this allpass filter gives us a way to get fractional sample delays in a discrete-time system, at least at low frequencies.

L.9.2 *In-lab section*

1. The lowpass filter we will use is a simple, length two moving average. If the input is x and the output is y, then the filter is given by the difference equation,

$$\forall\, n \in \textit{Integers}, \quad y(n) = 0.5(x(n) + x(n-1)). \tag{L.13}$$

   Find an expression for the frequency response of the lowpass filter given by (L.13). Use MATLAB to plot the magnitude and phase response over the frequency range 0 to π radians/sample. Is this a linear phase filter? If so, what is its delay?
2. In part 4 of the previous lab, you initialized a comb filter with random noise and produced a sound that reasonably well approximates a plucked string instrument, such as a guitar. We can improve the sound.

   Real instrument sounds have more dynamics in their frequency structure. That is, the spectrum of the sound within the first few milliseconds of plucking the string is different from the spectrum a second or so later. Physically, this is because the high frequency vibrations of the string die out more rapidly than the low frequency vibrations.

   We can approximate this effect by modifying the comb filter by inserting the lowpass filter given by (L.13) into the feedback loop. This can be accomplished by realizing the following difference equation:

$$\forall\, n \in \textit{Integers}, \quad y(n) = x(n) + 0.5\alpha(y(n-N) + y(n-N-1)).$$

   Modify your Simulink model you constructed in part 4 of the previous lab so that it uses a lowpass filter in the feedback loop, implementing this difference equation. Listen to the resulting sound, and compare it against the sound from the previous lab. Use $\alpha = 0.99$ and $N = 40$, as before. Can you hear the improvement?

3. In the last lab, you found that the tone of the sound generated by the comb filter had frequency 8,000/N, where N was the delay in the feedback loop, and 8,000 was the sampling frequency. You used $N = 40$ to get a fundamental frequency of 200 Hz. Now, you have added an additional lowpass filter, which introduces additional delay in the feedback loop. You have determined that additional delay in part 1 above. What is the fundamental frequency now?

   The comb filter delay can only delay by an integer number of samples. The lowpass filter introduces a fixed delay. Consequently, there are only certain fundamental frequencies that are possible. In particular, assuming the sample rate is 8,000 samples/second, is it possible to achieve a fundamental frequency of 440 Hz? This would be essential to have a reasonable guitar model, since we would certainly want to be able to play the note A-440 on the guitar. Determine the closest achievable frequency to 440 Hz. Is it close enough? In the independent section of this lab, you will show how to achieve a fundamental frequency very close to 440 Hz.

L.9.3 *Independent section*

1. Show analytically that the lowpass filter given by (L.13) has linear phase over the range of frequencies 0 to π radians/sample, and determine the slope. Verify that this agrees with the plot you constructed in the in-lab section.
2. In part 2 of the in-lab section, you combined an N-sample delay with a lowpass filter in the feedback path of a comb filter. Calculate the frequency response of this version of the comb filter, and plot its magnitude using MATLAB over the frequency range 0 to π. Compare it to the frequency response you calculated for the original comb filter in the previous lab. Find the fundamental frequency of the musical note from this plot and compare it to the answer that you gave in part 3 of the in-lab portion. Hint: The spectral peaks are very sharp, so you will need to calculate the magnitude frequency at many points in the range 0 to π to be sure to hit the peaks. We recommend calculating at least 2,000 points.
3. The reason that the comb filter with a lowpass filter in the feedback loop yields a much better plucked string sound than the comb filter by itself is that it more accurately models the physical phenomenon that higher frequency vibrations on the string die out faster than lower frequency vibrations. Plot the spectrogram using `specgram` of the generated sound to demonstrate this phenomenon, and explain how your spectrogram demonstrates it.
4. Verify that the frequency response (L.10) of the allpass filter has constant magnitude and linear phase for low frequencies by plotting it using MATLAB. Plot it for the following values of delay: $d = 0.1, 0.4, 0.7, and 1.0$. Plot it over the range of frequencies 0 to π radians/sample. Discuss how your plots support the conclusions about this filter. Hint: Use (L.12) to find a given d.
5. You determined in part 3 of the in-lab section that you could not get very close to A-440 with a comb filter with a lowpass filter in the feedback loop.

The allpass filter given by (L.10), however, can be used to achieve delays that are a fraction of a sample period. Implement the allpass filter, modifying your Karplus-Strong plucked string model by putting the allpass filter in the feedback loop. Set the parameters of the allpass filter and N to get an A-440. Show your Simulink diagram, and give the parameters of all the blocks.

Instructor Verification Sheet for Lab L.9

Name: _____ **Date:** _____

1. Magnitude and phase of lowpass filter. Linear phase? Delay?

   Instructor verification: _____

2. Improved plucked string sound.

   Instructor verification: _____

3. Fundamental frequencies that are possible.

   Instructor verification: _____

L.10 Modulation and demodulation

The purpose of this lab is to learn to use frequency domain concepts in practical applications. The application selected here is **amplitude modulation** (**AM**), a widely used technique in communication systems, including, of course, AM radio, but also almost all digital communication systems, including digital cellular telephones, voiceband data modems, and wireless networking devices. A secondary purpose of this lab is to gain a working knowledge of the **fast Fourier transform** (**FFT**) algorithm, and an introductory working knowledge of **filter design**. Note that this lab requires the Signal Processing Toolbox of MATLAB for filter design.

L.10.1 Background

Amplitude modulation

The problem addressed by AM modulation is that we wish to convey a signal from one point in physical space to another through some **channel**. The channel has certain constraints, and in particular, can typically only pass frequencies within a certain range. An AM radio station, for example, might be constrained by government regulators to broadcast only radio signals in the range of 720 to 760 kHz, a **bandwidth** of 40 kHz.

The problem, of course, is that the radio station has no interest in broadcasting signals that only contain frequencies in the range 720 to 760 kHz. They are more likely to want to transmit a voice signal, for example, which contains frequencies in the range of about 50 Hz to about 10 kHz. AM modulation deals with this mismatch by **modulating** the voice signal so that it is shifted to the appropriate frequency range. A radio receiver, of course, must **demodulate** the signal, since 720 kHz is well above the hearing range of humans.

In this lab, we present a somewhat artificial scenario in order to maximize the experience. We will keep all frequencies that we work with within the audio range so that you can listen to any signal. Therefore, we will not modulate a signal up to 720 kHz (you would not be able to hear the result). Instead, we present the following scenario:

> Assume we have a signal that contains frequencies in the range of about 100 to 300 Hz, and we have a channel that can pass frequencies from 700 to 1,300 Hz.* Our task will be to modulate the first signal so that it lies entirely within the channel **passband**, and then to demodulate to recover the original signal.

AM modulation is studied in detail in exercise 16 of chapter 10. In that problem, you showed that if

* Since Fourier transforms of real signals are symmetric, the signal also contains frequencies in the range −100 to −300 Hz, and the channel passes frequencies in the range −700 to −1,300 Hz.

$$y(t) = x(t)\cos(\omega_c t),$$

then the CTFT is

$$Y(\omega) = X(\omega - \omega_c)/2 + X(\omega + \omega_c)/2.$$

In this lab, we will get a similar result experimentally, but working entirely with discrete-time signals, and staying entirely within the audio range so that we can hear (and not just plot) the results.

The FFT algorithm

In order to understand AM modulation, we need to be able to calculate and examine Fourier transforms. We will do this numerically in this lab, unlike exercise 16 of chapter 10, where it is done analytically.

In lab L.7, we used a supplied function called `fourierSeries` to calculate the Fourier series coefficients A_k and ϕ_k for signals. In this lab, we will use the built-in function `fft`, which is used, in fact, by the `fourierSeries` function. Learning to use the FFT is extremely valuable; it is widely used in all analytical fields that involve time series analysis, including not just all of engineering, but also the natural sciences and social sciences. The FFT is also widely abused by practitioners who do not really understand what it does.

The FFT algorithm operates most efficiently on finite signals whose lengths are a power of 2. Thus, in this lab, we will work with what might seem like a peculiar signal length, 8,192. This is 2^{13}. At an 8,000 samples/second sample rate, it corresponds to slightly more than one second of sound.

Recall that a periodic discrete-time signal with period p has a discrete-time Fourier series expansion

$$x(n) = A_0 + \sum_{k=1}^{(p-1)/2} A_k \cos(k\omega_0 n + \phi_k) \tag{L.14}$$

for p odd and

$$x(n) = A_0 + \sum_{k=1}^{p/2} A_k \cos(k\omega_0 n + \phi_k) \tag{L.15}$$

for p even, where $\omega_0 = 2\pi/p$, the fundamental frequency in cycles/sample. Recall further that we can alternatively write the Fourier series expansion in terms of complex exponentials,

$$x(n) = \sum_{k=0}^{p} X_k e^{ik\omega_0 n}. \tag{L.16}$$

Note that this sum covers one cycle of the periodic signal. If what we have is a finite signal instead of a periodic signal, then we can interpret the finite signal as being one cycle of a periodic signal.

In chapter 10, we describe four Fourier transforms. The only one of these that is computable on a computer is the **DFT**, or **discrete Fourier transform**. For a periodic discrete-time signal x with period p, we have the **inverse DFT**, which takes us from the frequency domain to the time domain,

$$\forall\, n \in \mathit{Integers}, \quad x(n) = \frac{1}{p} \sum_{k=0}^{p-1} X'_k e^{ik\omega_0 n}, \tag{L.17}$$

and the **DFT**, which takes us from the time domain to the frequency domain,

$$\forall\, k \in \mathit{Integers}, \quad X'_k = \sum_{m=0}^{p-1} x(m) e^{-imk\omega_0}. \tag{L.18}$$

Comparing (L.17) and (L.16), we see that the DFT yields coefficients that are just scaled versions of the Fourier series coefficients. This scaling is conventional.

In lab L.7, we calculated A_k and ϕ_k. In this lab, we will calculate X'_k. This can be done using (L.18). The FFT algorithm is simply a computationally efficient algorithm for calculating (L.18).

In MATLAB, you will collect 8,192 samples of a signal into a vector and then invoke the `fft` function. Invoke `help fft` to verify that this function is the right one to use. If your 8,192 samples are in a vector `x`, then `fft(x)` will return a vector with 8,192 complex numbers, representing $X_0, \ldots, X_{8,191}$.

From (L.18) it is easy to verify that $X_k = X_{k+p}$ for all integers k (see part 1 of the following in-lab section). Thus, the DFT X is a periodic, discrete function with period p. If you have the vector `fft(x)`, representing $X_0, \ldots, X_{8,191}$, you know all X_k. For example,

$$X_{-1} = X_{-1+8,192} = X_{8,191}.$$

From L.17, you can see that each X_k scales a complex exponential component at frequency $k\omega_0 = k2\pi/p$, which has units of samples/second. In order to interpret the DFT coefficients X_k, you will probably want to convert the frequency units to Hertz. If the sampling frequency is f_s samples/second, then you can do the conversion as follows (see Basics: Frequencies in Hertz and radians on page 248 of the text):

$$\frac{(k2\pi/p)\,[\text{radians/sample}] f_s\,[\text{samples/second}]}{2\pi\,[\text{radians/cycle}]} = kf_s/p\,[\text{cycles/second}] \tag{L.19}$$

Thus, each X_k gives the DFT value at frequency kf_s/p Hz. For our choices of numbers, $f_s = 8{,}000$ and $p = 8{,}192$, so X_k gives the DFT value at frequency $k \times 0.9766$ Hz.

Filtering in MATLAB

The `filter` function can compute the output of an LTI system given by a difference equation of the form

$$\begin{aligned} a_1 y(n) = b_1 x(n) + b_2 x(n-1) + \cdots + b_N x(n-N+1) \\ - a_2 y(n-1) - \cdots - a_M y(n-M+1). \end{aligned} \tag{L.20}$$

To find the output y, first collect the (finite) signal x into a vector. Then collect the coefficients $a_1, \ldots, a_N$ into a vector `A` of length N, and the coefficients $b_1, \ldots, b_M$ into a vector `B` of length M. Then just do

```
y = filter(B, A, x);
```

Example L.1: Consider the difference equation

$$y(n) = x(n) - 0.95y(n-1).$$

We can find and plot the first 100 samples of the impulse response by letting the vector x be an impulse and using `filter` to calculate the output:

```
x = [1, zeros(1,99)];
y = filter([1], [1, 0.95], x);
stem(y);
```

which yields the plot shown in figure L.12. The natural question that arises next is how to decide on the values of `B` and `A`. This is addressed in the next section. ❐

Filter design in MATLAB

The signal processing toolbox of MATLAB provides a set of useful functions that return filter coefficients `A` and `B` given a specification of a desired frequency response. For example, suppose we have a signal sampled at 8 kHz and we wish to design a filter that passes all frequency components below 1 kHz and removes all frequency components above that. The following MATLAB command designs a filter that approximates this specification:

```
[B, A] = butter(10, 0.25);
```

The first argument, called the **filter order**, gives M and N in (L.20) (a constraint of the `butter` function is that $M = N$). The second argument gives the **cutoff frequency** of the filter as a fraction of half the sampling frequency. Thus, in the above, the cutoff frequency is $0.25 * (8{,}000/2) = 1{,}000$ Hertz. The cutoff frequency

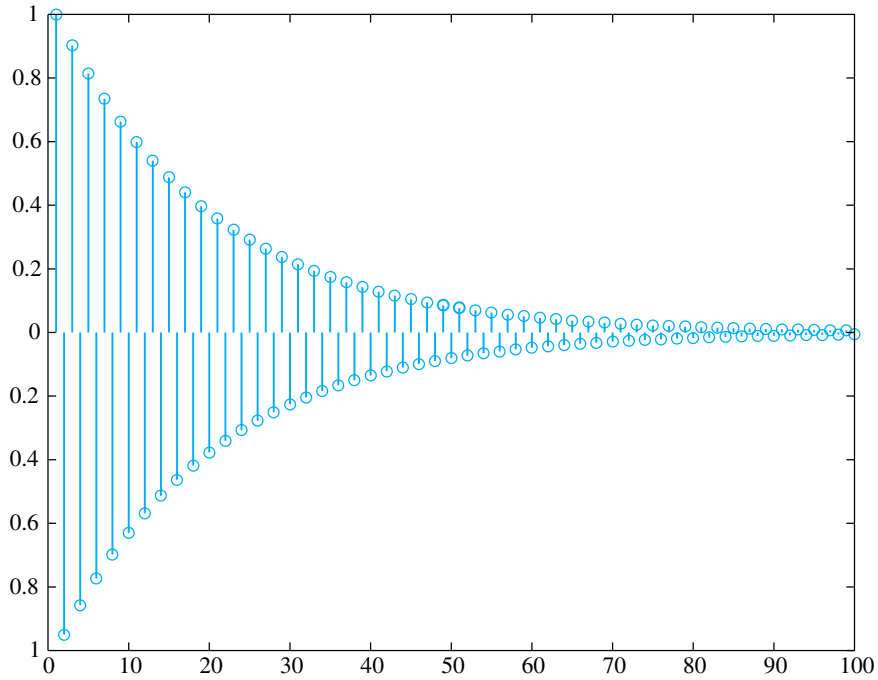


FIGURE L.12: Impulse response of a simple filter.

is by definition the point at which the magnitude response is $1/\sqrt{2}$. The returned arrays `B` and `A` are the arguments to supply to `filter` to calculate the filter output.

The frequency response of this filter can be plotted using the `freqz` function as follows:

```
[H,W] = freqz(B,A,512);
plot(W*(4000/pi), abs(H));
xlabel('frequency');
ylabel('magnitude response');
```

which yields the plot shown in figure L.13. (The argument 512 specifies how many samples of the continuous frequency response we wish to calculate.)

This frequency response bears further study. Notice that the response transitions gradually from the passband to the stopband. An abrupt transition is not implementable. The width of the transition band can be reduced by using an order higher than 10 in the `butter` function, or by designing more sophisticated filters using the `cheby1`, `cheby2`, or `ellip` functions in the signal processing toolbox. The Butterworth filter returned by `butter`, however, will be adequate for our purposes in this lab.

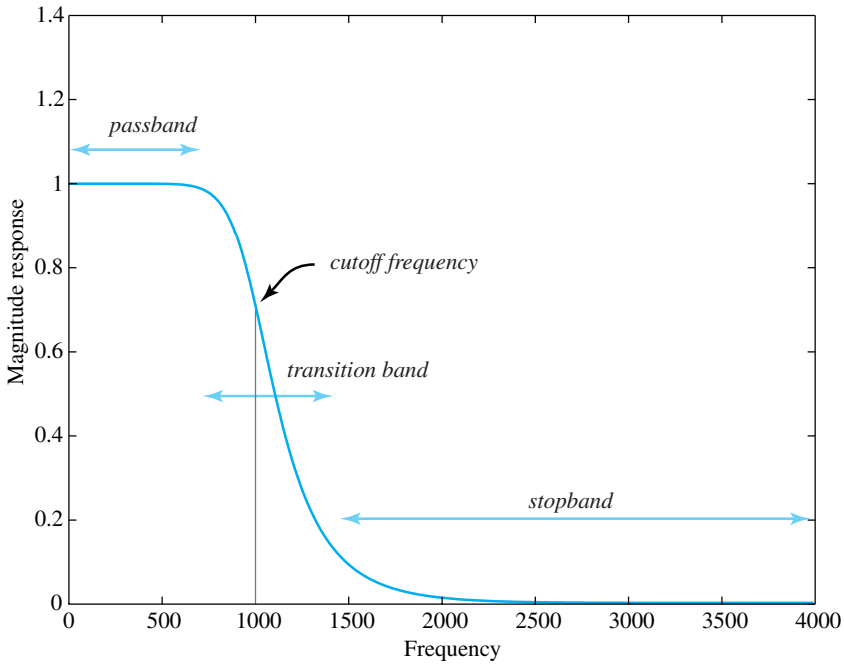


FIGURE L.13: Frequency response of a 10th order Butterworth lowpass filter.

Using a higher order to get a narrower transition band can be an expensive proposition. The function `filter` works by implementing the difference equation (L.20). As M and N get larger, each output sample $y(n)$ requires more computation.

The first 50 samples of the impulse response of the filter can be plotted using the following MATLAB code:

```
x = [1, zeros(1,49)];
y = filter(B, A, x);
stem(y);
```

This yields the plot shown in figure L.14.

L.10.2 *In-lab section*

1. Use (L.18) to show that $X'_k = X'_{k+p}$ for all integers k. Also, show that the DFT is conjugate symmetric (i.e., $X'_k = (X'_{-k})^*$ for all integers k, assuming $x(n)$ is real for all integers n).
2. In part 2 of the in-lab portion of lab L.7, we studied a **chirp** signal. We use a similar signal here, although it varies over a narrower range of frequencies. Construct the signal `x` defined by:

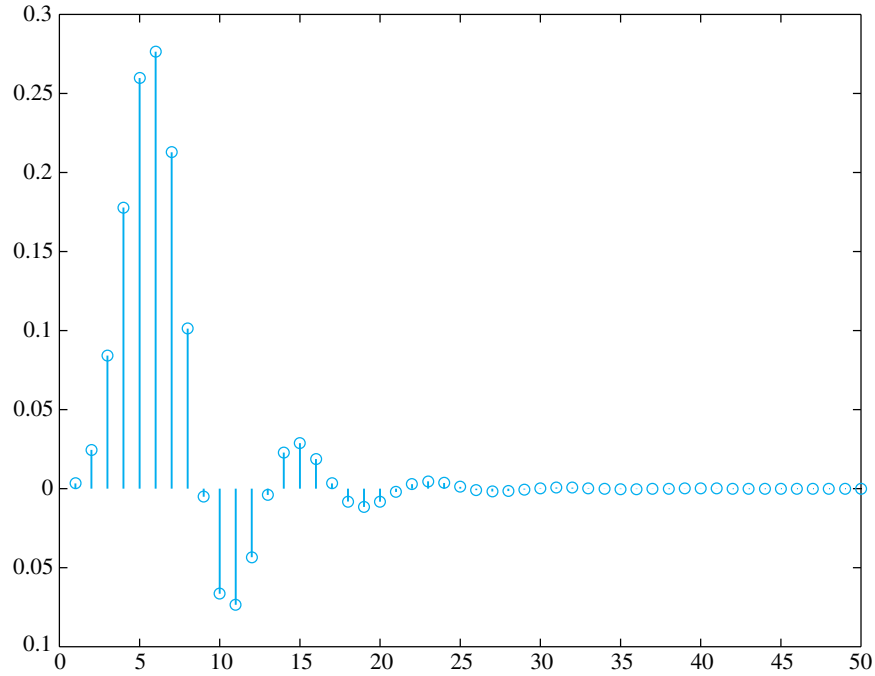


FIGURE L.14: Impulse response of a 10th order Butterworth lowpass filter.

```
t = [0:1/8000:8191/8000];
x = sin(2*pi*100*t + 2*pi*100*(t.*t));
```

This is a chirp that varies from about 100 Hz to about 300 Hz. Listen to it. Calculate its DFT using the `fft` function, and plot the magnitude of the DFT. Construct your plot in such a way that your horizontal axis is labeled in Hertz, not in the index k of X_k. The horizontal axis should vary in frequency from 0 to 8,000 Hz.

3. Your plot from part 2 should show frequency components in the range 100 Hz to 300 Hz, but in addition, it shows frequency components in the range 7,700 to 7,900. These extra components are the potentially the most confusing aspect of the DFT, but in fact, they are completely predictable from the mathematical properties of the DFT.

   Recall that the DFT of a real signal is conjugate symmetric. Hence,

$$|X'_k| = |X'_{-k}|.$$

   Thus, if there are frequency components in the range 100 to 300 Hz, then there should also be frequency components with the same magnitude in the

range −100 to −300 Hz. These do not show up on your plot simply because you have not plotted the frequency components at negative frequencies.

Recall that the DFT is periodic with period p. That is, $X_k = X_{k+p}$ for all integers k. Recall from (L.19) that the $k-th$ DFT coefficient represents a frequency component at kf_s/p Hertz, where f_s is the sampling frequency 8,000 Hertz. Thus, a frequency component at some frequency f must be identical to a frequency component at $f + f_s$. Therefore, the components in the range −100 to −300 Hertz must be identical to the components in the range 7,700 to 7,900 Hertz! The image we are seeing in that latter range is a consequence of the conjugate symmetry and periodicity of the DFT!

Since the DFT is periodic with period 8,000 Hertz (when using units of Hertz), it possibly makes more sense to plot its values in the range −4,000 to 4,000 Hertz, rather than 0 to 8,000 Hertz. This way, we can see the symmetry. Since the DFT gives the weights of complex exponential components, the symmetry is intuitive, because it takes two complex exponentials with frequencies that are negatives of one another to yield a real-valued sinusoid.

Manipulate the result of the `fft` function to yield a plot of the DFT of the chirp where the horizontal axis is −4,000 to 4,000 Hertz. It is not essential to include both endpoints, at −4,000 and at 4,000 Hertz, since they are constrained to be identical anyway by periodicity.

L.10.3 *Independent section*

1. For the chirp signal as above, multiply it by a sine wave with frequency 1 kHz, and plot the magnitude of the DFT of the result over the frequency range −4,000 to 4,000 Hz. Verify that the resulting signal will get through the channel described in the scenario on page 65. Listen to the modulated chirp. Does what you hear correspond with what you see in the DFT plot?
2. The modulated chirp signal constructed in the previous part can be demodulated by multiplying it again by a sine wave with frequency 1 kHz. Form that product, and plot the magnitude of the DFT of the result over the frequency range −4,000 to 4,000 Hz. How does the result differ from the original chirp? Listen to the resulting signal. Would this be an acceptable demodulation by itself?
3. Use the `butter` function to design a filter that will process the result of the previous part so that it more closely resembles the original signal. You should be able to get very close with a modest filter order (say, 5). Filter the result of the previous part, listen to the result, and plot the magnitude of its DFT in the frequency range −4,000 to 4,000 Hz.

   The modulation and demodulation method you have just implemented is similar to what is used many communication systems. A number of practical problems have to be overcome in practice, however. For example, the receiver usually does not know the exact frequency and phase of the carrier signal, and hence it has to somehow infer this frequency and phase from the

signal itself. One technique is to simply include the carrier in the modulated signal by adding it in. Instead of transmitting

$$y(t) = x(t)\cos(\omega_c t),$$

we can transmit

$$y(t) = (1 + x(t))\cos(\omega_c t),$$

in which case, the transmitted signal will contain the carrier itself. This can be used for demodulation. Another technique is to construct a **phase locked loop**, a clever device that extracts the carrier from the transmitted signal without it being explicitly present. This method is used in digital transmission schemes. The details, however, must be left to a more advanced text.

In the scheme we have just implemented, the amplitude of a carrier wave is modulated to carry a signal. It turns out that we can also vary the phase of the carrier to carry additional information. Such **AM-PM** methods are used extensively in digital transmission. These methods make more efficient use of precious radio bandwidth than AM alone.

Instructor Verification Sheet for Lab L.10

Name: _____ **Date:** _____

1. Verify periodicity and conjugate symmetry of the DFT.

   Instructor verification: _____

2. Plot the magnitude of the DFT, correctly labeled, from 0 to 8,000 Hz.

   Instructor verification: _____

3. Plot of the magnitude of the DFT, correctly labeled, from –4,000 to 8,000 Hz.

   Instructor verification: _____

L.11 Sampling and aliasing

The purpose of this lab is to study the relationship between discrete-time and continuous-time signals by examining sampling and aliasing. Of course, a computer cannot directly deal with continuous-time signals. So instead, we will construct discrete-time signals that are defined as samples of continuous-time signals, and then operate entirely on them, downsampling them to get new signals with lower sample rates, and upsampling them to get signals with higher sample rates.

Note that this lab has no independent part. Therefore, no lab writeup needs to be turned in. The instructor verification sheet is sufficient.

L.11.1 *In-lab section*

1. Recall from lab L.7 that a **chirp** signal given by

   $$x(t) = \sin(2\pi ft^2)$$

   has **instantaneous frequency**

   $$\frac{d}{dt} 2\pi ft^2 = 4\pi ft$$

   in radians per second, or

   $$2ft$$

   in Hertz. A sampled signal $y = Sampler_T(x)$ with sampling interval T will be

   $$y(n) = \sin(2\pi f(nT)^2).$$

   Create in MATLAB a chirp sampled at 8,000 samples/second that lasts 10 seconds and sweeps from frequency 0 to 12 kHz. Listen to the chirp. Explain the aliasing artifacts that you hear.
2. For the remainder of this lab, we will work with a particular chirp signal that has a convenient Fourier transform for visualizing and hearing aliasing effects. For efficient computation using the FFT algorithm, we will work with $8{,}192 = 2^{13}$ samples, giving slightly more than 1 second of sound at an 8,000 samples/second sample rate. You may wish to review lab L.10, which explains how to create well-labeled plots of the DFT of a finite signal using the FFT algorithm.

   The chirp signal we wish to create is given by

   $$\forall\, t \in [0, D], \quad x(t) = f(t)\sin(2\pi ft^2)$$

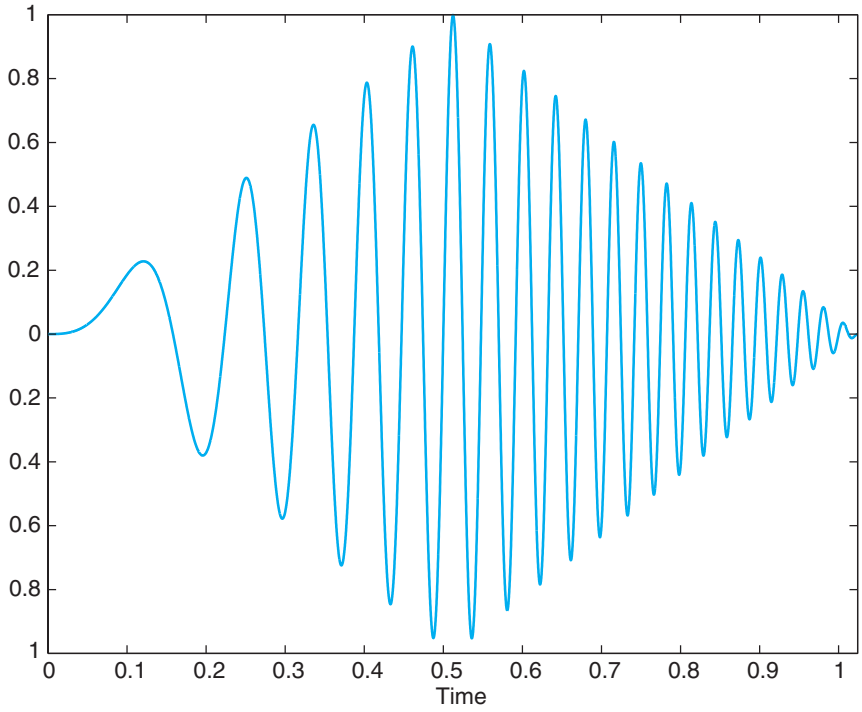


FIGURE L.15: Chirp signal with a triangular envelope.

where D is the duration of the signal and $f(t)$ is a time-varying amplitude given by

$$f(t) = 1 - |t - D/2|/(D/2).$$

This chirp starts with amplitude 0, rising linearly to peak at the midpoint of the duration, and then falls again back to zero, as shown in figure L.15. Thus, it has a triangular envelope.

Generate such a chirp that lasts for 8,192 samples at an 8-kHz sample rate and sweeps from a frequency of zero to 2,500 Hz. Listen to the resulting sound. Are there any aliasing artifacts? Why or why not?

3. Use the FFT algorithm, as done in lab L.10, to create a plot of the magnitude of the DFT of the chirp signal from the previous part over the frequency range −4 kHz to 4 kHz. Make sure the horizontal axis of your plot is labeled in Hertz. Does your plot look like a sensible frequency-domain representation of this chirp?
4. Modify your chirp so that it sweeps from 0 to 5 kHz. Listen to it. Do you hear aliasing artifacts? Plot the magnitude of the DFT over −4 kHz to 4 kHz. Can

you see the aliasing artifacts in this plot? Explain why the plot has the shape that it does.

5. Return to the chirp that you generated in part 2, which sweeps from 0 to 2,500 Hz. Create a new signal with sample rate 4,000 samples/second by **downsampling** that chirp. That is, create a vector in MATLAB that has half the length by selecting every second sample from the original chirp. For example, if $y(n)$ is the original chirp, define w by

$$w(n) = y(2n).$$

Now plot the magnitude DFT of this signal.* Since the sampling rate is lower by a factor of 2, you should plot over the frequency interval -2 kHz to 2 kHz. Is there aliasing distortion? Why?

6. Return again to the chirp that you generated in part 2, which sweeps from 0 to 2,500 Hz. Create a new signal with sample rate 16,000 samples/second by **upsampling** that chirp. That is, create a vector in MATLAB that has twice the length by inserting zero-valued samples between each pair of samples from the original chirp. For example, if $y(n)$ is the original chirp, define z by

$$z(n) = \begin{cases} y(n/2) & \text{if } n \text{ is even} \\ 0 & \text{otherwise.} \end{cases}$$

Now plot the magnitude DFT of this signal. Since the sampling rate is higher by a factor of 2, you should plot over the frequency interval -8 kHz to 8 kHz. Explain this plot. Listen to the sound by giving a sampling frequency argument to `soundsc` as follows:†

```
soundsc(w, 16000);
```

How does the sound correspond with the plot?

7. Design a Butterworth filter using the `butter` function in MATLAB to get back a high quality rendition of the original chirp, but with a sample rate of 16,000 Hz. Filter the signal with your filter and listen to it.

Note that this technique is used in most compact disc players today. The signal on the CD is sampled at 44,100 samples/second. The CD player first upsamples it by a factor of 4 or 8 to get a sample rate of 176,400 samples/second or 352,800 samples/second. The upsampling is accomplished by inserting zero-valued samples. The resulting signal is then digitally filtered to get a

*Unfortunately, MATLAB does not document what actually happens when you give a sampling frequency of 4,000 to the `sound` or `soundsc` functions. On at least some computers, the sound that results from attempting to do this is difficult to interpret. Thus, we do not recommend attempting to listen to this downsampled chirp.

†The audio hardware on many computers directly supports a sample rate of 16,000 samples/second, so at least on such computers, MATLAB seems to correctly handle this command.

high sample rate and accurate rendition of the audio. The higher sample rate signal is then converted to a continuous-time signal by a digital to analog converter that operates at the higher sample rate. This technique shifts most of the difficulty of rendering a high-quality audio signal to the discrete-time domain, where it can be done with digital circuits or microprocessors (such as programmable DSPs) rather than with analog circuits.

Instructor Verification Sheet for Lab L.11

Name: _____ **Date:** _____

1. Explain the aliasing artifacts that you hear.

   Instructor verification: _____

2. Generate chirp with triangular envelope.

   Instructor verification: _____

3. Generate a frequency-domain plot of the chirp with a triangular envelope.

   Instructor verification: _____

4. Show a frequency-domain plot with aliasing distortion and explain the distortion.

   Instructor verification: _____

5. Show a frequency-domain plot with double chirp and explain the sound.

   Instructor verification: _____

6. Give a reasonable filter design.

   Instructor verification: _____