OPEN SIGNALS AND SYSTEMS LABORATORY EXERCISES



Andrew K. Bolstad & Julie A. Dickerson
Iowa State University

LibreTexts™

Open Signals and Systems Laboratory Exercises (Bolstad and Dickerson)

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (https://LibreTexts.org) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide an accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden born from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.



LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by NICE CXone Expert and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptions contact info@LibreTexts.org or visit our main website at https://LibreTexts.org.

This text was compiled on 05/13/2026

TABLE OF CONTENTS

Licensing

Acknowledgments

Introduction

1: Labs

- 1.1: MATLAB Basics
- 1.2: Tuning Forks
- 1.3: Impulse Response and Auralization
- 1.4: Frequency Response - Notch and Bandpass Filters
- 1.5: Fourier Series
- 1.6: Introduction to Digital Images
- 1.7: Filtering Digital Images

Detailed Licensing

Detailed Licensing

Licensing

*A detailed breakdown of this resource's licensing can be found in **Back Matter/Detailed Licensing**.*

Acknowledgments

Introduction

Open Signals and Systems Laboratory Exercises is a collection of lab assignments that have been used in EE 224: Signals and Systems I in the Department of Electrical and Computer Engineering at Iowa State University. These lab exercises have been curated, edited, and presented in a consistent format to improve student learning.

The first lab exercise introduces students to the MATLAB environment and language with an emphasis on signal processing applications. Students solve a system of equations, manipulate complex numbers, vectorize an equation, and manipulate sound and image files. The next lab exercise reinforces the concepts of period and frequency by having students record the sound of a tuning fork and measuring characteristics of the audio signal. The third lab uses the impulse response of a system to simulate how things sound in different locations.

Labs four and five give students practice with frequency domain signal manipulation. Lab four explores various filter types, including notch and bandpass. Lab five has students compare the Fourier series coefficients of a tone played by a trumpet and a flute. Students also have the opportunity to make their own \instrument" by manipulating Fourier series coefficients. These lab exercises are somewhat long; we often give students two weeks to complete each of them.

The last two lab exercises in this collection focus on image processing. Lab 6 introduces students to basic pixel manipulation and displaying images in MATLAB. Lab 7 lets students apply two-dimensional filters for noise reduction and edge detection.

With a few exceptions noted above, these labs can usually be completed in a two hour lab session with students working in pairs. The collection presented here represents close to a full semester of lab exercises; instructors may wish to supplement this collection with labs covering modulation and sampling, or require a student project.

CHAPTER OVERVIEW

1: Labs

1.1: MATLAB Basics
1.2: Tuning Forks
1.3: Impulse Response and Auralization
1.4: Frequency Response - Notch and Bandpass Filters
1.5: Fourier Series
1.6: Introduction to Digital Images
1.7: Filtering Digital Images

This page titled 1: Labs is shared under a not declared license and was authored, remixed, and/or curated by Andrew K. Bolstad & Julie A. Dickerson (Iowa State University Digital Press) .

1.1: MATLAB Basics

Learning Objectives

By the end of this lab, students will be able to:

1. Get help and documentation for MATLAB functions
2. Solve matrix-vector equations in MATLAB
3. Manipulate complex numbers in MATLAB
4. Vectorize a simple equation
5. Use MATLAB's plotting tools to graph signals

This lab will also preview future labs by demonstrating how to play audio signals and display images.

Overview

This lab introduces the MATLAB computing environment. MATLAB is an extremely useful tool for signals and systems as well as for many other computing tasks. Unlike C, C++, or Java, there is no need to compile MATLAB code, making debugging and experimenting much easier. The purpose of this lab is not to provide a comprehensive introduction to MATLAB, but to help students get comfortable enough to learn more on their own. Future labs will introduce new MATLAB content as necessary.

Note: Most of this lab was developed using MATLAB versions R2017b and R2019a. Compatibility across versions is generally very good in MATLAB, but the appearance of the GUIs tends to change.

Pre-Lab Reading

MATLAB is a commercial software product produced by MathWorks. The software includes both a numeric computing environment and a programming language, though "MATLAB" often refers to the programming language. MATLAB is commonplace in industry and academia. You will benefit from understanding the basics of using MATLAB.

Many university students can obtain student versions of MATLAB for free. Instructions for Iowa State University students can be found here: https://it.engineering.iastate.edu/how-to/installing-matlab/. Free alternatives to MATLAB include GNU Octave (https://www.gnu.org/software/octave/), Scilab (https://www.scilab.org/), and FreeMat(http://freemat.sourceforge.net/).

MATLAB stands for "MATrix LABoratory," as it was first developed to allow university students to easily use numeric computing software for linear algebra. Part of what makes MATLAB easy to use is that there is no need to compile MATLAB code before running it. Code can be entered and executed line by line in the Command Window. Reflecting its roots in linear algebra[1], all variables are essentially matrices by default. Variables do not have to be declared before they are used, and usually it is unnecessary to specify the class of a variable. These factors make it easy to get started in MATLAB, though they can be disorienting for those used to compiled languages like C, C++, and Java.

Lab Exercises

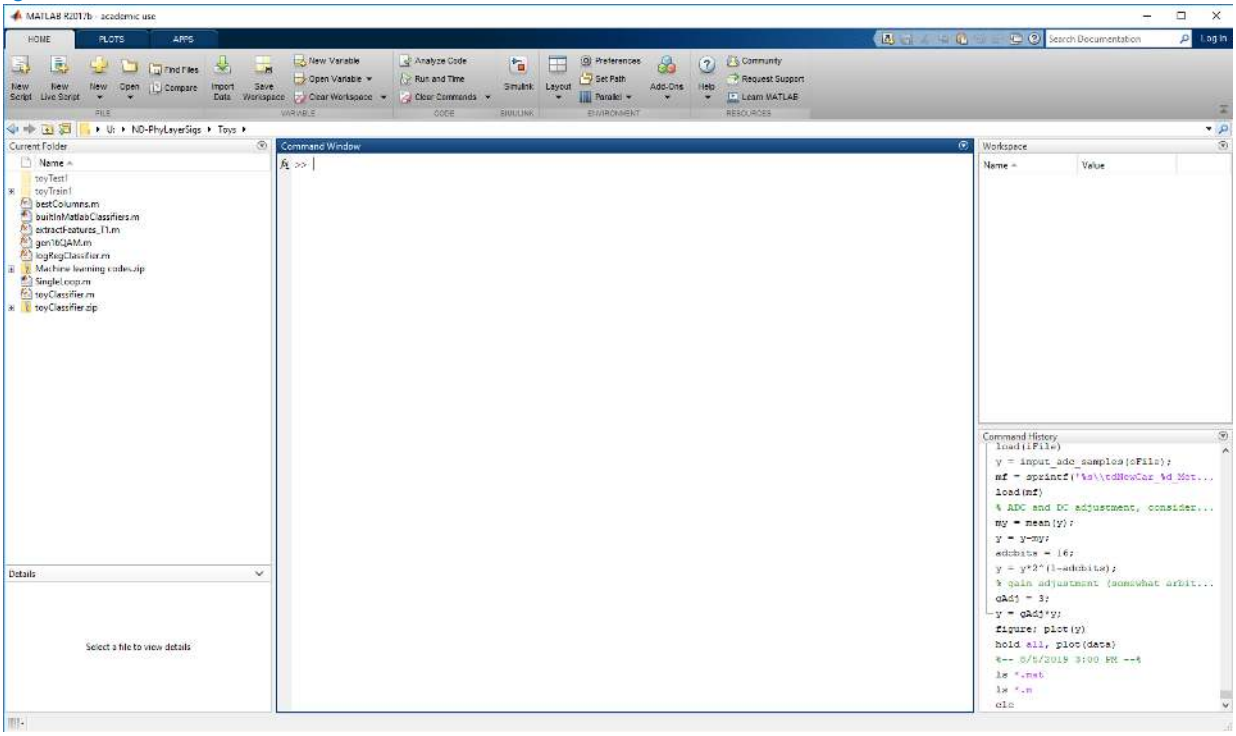


Figure 1.1.1: The MATLAB Desktop

Upon starting up MATLAB, you will see the MATLAB Desktop (see Figure 1.1.1) which consists of the Tool Strip, Current Folder window, Command Window, Workspace, and Command History. The purpose of these windows is largely self-explanatory, but we will touch on them occasionally as necessary. Perhaps the best way to jump into MATLAB is to learn how to get help. You can simply type `help` followed by the name of a function to get help on that function. You can even get help for the function help:

```
>> help help
```

Another great way to get help is with the doc command. Whereas `help` gives you a (usually) brief explanation in the Command Window, `doc` opens up more thorough documentation in a new window. Use the doc command to find out about the many ways to use the colon (:) in MATLAB:

```
>> doc colon
```

MATLAB also has built in demos to help you use various functions and tools. You can type `help demo` to learn about how to run the examples, or you can just type `demo` to open up a new window with featured examples. Find and click on the “Basic Matrix Operations” demo. This will display the demo in the help window. Click the “Open Live Script” button in the upper right. This will open the demo as a live script in MATLAB’s Live Editor. Step through the live script by clicking the “Run Section” or “Run and Advance” buttons in the Live Editor’s Tool Strip. Make sure you understand what is happening with each command. (Most of you probably don’t know what it means to “convolve” two vectors yet, but you will in a few weeks!)

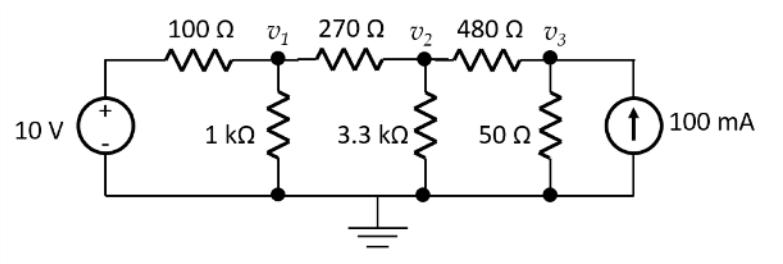


Figure 1.1.2: A simple circuit

Now that you have seen an example of solving a matrix equation $\mathbf{y} = \mathbf{H}\mathbf{x}$ for $\mathbf{x}$, let's apply this to something you know. Consider the circuit shown in Figure 1.1.2. In EE 201, you learned that the voltages v_1, v_2, and v_3 in this circuit can be found via the Node Voltage Method. Assuming you remember how to do that, you will arrive at the following set of equations:

$$\begin{aligned}\frac{v_1 - 10}{100} + \frac{v_1}{1000} + \frac{v_1 - v_2}{270} &= 0\\ \frac{v_2 - v_1}{270} + \frac{v_2}{3300} + \frac{v_2 - v_3}{480} &= 0\\ \frac{v_3 - v_2}{480} + \frac{v_3}{50} &= 0.1\end{aligned}$$

We can rewrite these as a matrix vector equation:

$$\begin{bmatrix} \frac{1}{100} + \frac{1}{1000} + \frac{1}{270} & \frac{-1}{270} & 0 \\ \frac{-1}{270} & \frac{1}{270} + \frac{1}{3300} + \frac{1}{480} & \frac{-1}{480} \\ 0 & \frac{-1}{480} & \frac{1}{480} + \frac{1}{50} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} \frac{10}{100} \\ 0 \\ 0.1 \end{bmatrix}$$

Use MATLAB to solve for the voltages v_1, v_2, and v_3. Be sure to include the code you used in your lab report.

HINT: You can use the diary function to keep track of the commands you use in MATLAB. Type `help diary` to find out how.

Complex Numbers in MATLAB

Consider the A/C circuit shown in Figure 1.1.3. Recall that the steady-state response of this circuit can be found using phasors. In the phasor domain, we can use a voltage divider to find the phasor voltage on the capacitor: $\mathbf{V}_C = \frac{-j\frac{100}{3}}{50 - j\frac{100}{3}} 5\angle 10^\circ$. Rearranging and recalling that the phasor notation $5\angle 10^\circ$ means $5e^{j10\frac{\pi}{180}}$, we have:

$$\mathbf{V}_C = \frac{1}{j1.5 + 1} 5e^{j\frac{\pi}{18}} \tag{1.1.1}$$

Let's use MATLAB to find $\mathbf{V}_C$.

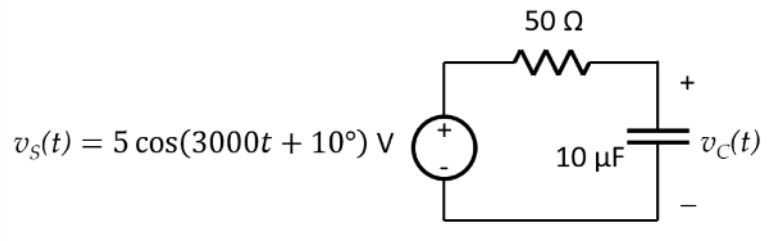


Figure 1.1.3: An RC circuit

MATLAB naturally handles complex numbers and complex arithmetic. There are several ways to represent the imaginary unit ($\sqrt{-1}$). Try typing the following in MATLAB.

```
>> a = i;
>> b = j;
>> c = 1i;
>> d = 1j;
>> a, b, c, d
```

You should see that MATLAB displays the value of each variable as 0.0000 + 1.0000i. You can check that this is indeed $\sqrt{-1}$ by using the equality operator "==":

```
>> a == sqrt(-1)
```

Notice the difference between assignment (=) and equality (==).

Now we can solve Equation [eq:phasor] in MATLAB. Rather than just typing in the right-hand side of the equation, we will use variables for each quantity:

```
>> om = 3000; C = 1e-5; R = 50;
>> Vs = 5*exp(1i*10*pi/180);
>> Zc = -1i/om/C;
>> Zr = R;
>> Vc = Zc/(Zr+Zc)*Vs
```

MATLAB should display an answer of 1.9158 - 2.0055i. Before we explore what that means, there are a few important things to notice about MATLAB in these lines. First, we can type lots of commands on one line; separating by a semicolon (;) stops MATLAB from displaying the result of each. (Try using a comma (,) instead.) Second, angles in MATLAB are assumed to be in radians, so we converted 10° to $10\pi/180$ radians. Third, the number π is built in to MATLAB. Other built-in values include infinity (`inf`), not-a-number (`NaN`), and the output of the last command (`ans`)[2]. Fourth, MATLAB uses order of operations the way most scientific calculators and other programming languages do. You can find out exactly how MATLAB orders operations by typing "operator precedence" in the search bar in the upper right corner of the MATLAB desktop.

Using MATLAB we found that $\mathbf{V}_C = 1.9158 - 2.0055i$ V; however, we would like to have the answer in phaser form. In MATLAB we can use `abs` and `angle` to convert $\mathbf{V}_C$ to polar form or phasor form. Try it yourself. Once you have the magnitude and phase of the capacitor voltage, find the time-domain voltage, $v_C(t)$, and record this in your lab report. Remember to use a degree sign ($^\circ$) if you choose to report the phase angle in degrees.

There are a few other commands that are useful when working with complex numbers. You can type `real(Vc)` or `imag(Vc)` to get ther real or imaginary parts of `Vc` . `conj(Vc)` returns the complex conjugate of `Vc` .

```
>> real(Vc)

ans =

    1.9158

>> imag(Vc)

ans =

   -2.0055

>> conj(Vc)

ans =

   1.9158 + 2.055i
```

Vectorization

MATLAB is optimized for operations involving matrices and vectors. For example, we could use the following code to multiply matrices $\mathbf{A}$ and $\mathbf{B}$ and store the result in matrix $\mathbf{C}$ (assuming $\mathbf{C}$ has been initialized to all zeros):

```
>> for x = 1:size(A,1)
>>     for y = 1:size(B,2)
>>         for z = 1:size(A,2)
>>             C(x,y) = C(x,y) + A(x,z) * B(z,y);
>>         end
>>     end
>> end
```

Not ony is this annoying to type, but it also executes less efficiently than simply typing:

```
>> C = A*B;
```

Likewise, matrix-vector products can be found by typing `y = H*x` for matrix $\mathbf{H}$ and vectors $\mathbf{x}$ and $\mathbf{y}$. We can multiply matrices or vectors by scalars by simply typing `a*H` or `a*y` . Almost all MATLAB operations are designed to be applied to matrices. In fact, it is good programming practice to avoid for loops in MATLAB (though sometimes they are unavoidable).

Let's use vectorization to our advantage. Suppose we want to know the steady-state output voltage $\mathbf{V}_C$ for the circuit in Figure 3 for the following input voltages:

- $v_S(t) = 5\cos\left(3000t + 10^\circ\right)$ V
- $v_S(t) = 5\cos\left(3000t + 20^\circ\right)$ V
- $v_S(t) = 10\cos\left(3000t + 25^\circ\right)$ V
- $v_S(t) = 15\cos\left(3000t\right)$ V

Since all the sources have the same frequency (3000 radians/s), we can convert these to phasors and vectorize Equation [eq:phasor]:

```
>> Vs = [5*exp(1i*10*pi/180); 5*exp(1i*20*pi/180); ...
10*exp(1i*25*pi/180); 15];
>> Vc = Zc/(Zr+Zc)*Vs
```

This creates a column vector of source voltages in phasor form, `Vs`, and a column vector of capacitor voltages in phasor form, `Vc`. (The ellipsis (...) allows you to write one MATLAB command on multiple lines.) We have solved all four circuit analysis problems at once! Use `abs` and `angle` to convert the answers to the time domain and record these in your report.

Working with Signals in MATLAB

Strictly speaking, MATLAB (and digital computers in general) can only operate on discrete-time, discrete-amplitude signals. Furthermore, digital computers can only store a finite number of signal values at a time. Recall that a signal is a function of one or more independent variables, and those independent variables are integers for discrete-time signals.

Consider the discrete-time signal $x[n] = \cos(2\pi 0.01n)$. We can represent this signal as a vector in MATLAB for the range of integers $n = 0, 1, 2, \ldots, 9$ as follows:

```
>> n = 0:9;
>> x = cos(2*pi*0.01*n)
```

The ten values stored in the array x should display as a row vector. We can change this to a column vector using an apostrophe (’):

```
>> y = x'
```

MATLAB indexes arrays starting with the number 1, not the the number 0 as many other programming languages do. (This can be confusing if you are used to C/C++ or Java.) Thus, typing `x(1)` in MATLAB will display 1.0000, which is equal to $\cos(2\pi 0.01(0))$, whereas typing `x(0)` will result in an error. As you saw in the documentation for the colon operator in Section 4.1, you can access various subsets of the vector `x` as follows:

```
>> x(4:7)

ans =

    0.9823    0.9686    0.9511    0.9298

>> x(1:2:end)

ans =

    1.0000    0.9921    0.9686    0.9298    0.8763

>> x(2:2:end)

ans =

    0.9980    0.9823    0.9511    0.9048    0.8443
```

These are the fourth through seventh, odd-numbered, and even-numbered elements of the vector `x`, respectively.

Plotting Signals

Often signals of interest are functions of time. In the discrete-time case, the integers that index a signal correspond to different points in time. Let’s create another signal:

```
>> t = -0.01:1/44100:0.01;
>> x = cos(2*pi*440*t); % Don't forget the semicolon!
```

The first line created an array called `t`, which we are using to store values of time in seconds. The second line created an array called `x`, which holds the values of our signal as a function of time.

MATLAB includes many built-in functions for plotting. We will start with the simplest: `plot`.

```
>> figure; plot(t,x)
```

The command `figure` opens up a new figure window. It is good practice to use `figure` before `plot`; otherwise the plot command will use the last figure window that was used and overwrite whatever was plotted there. The plot command plots the values in array `x` as a function of the values in array `t`. What happens if you omit `t`? Type the following to find out:

```
>> figure; plot(x)
```

Describe what happens in your lab report.

Let's recreate the first plot and add some labels.

```
>> figure; plot(t,x)
>> xlabel(`time (s)')
>> ylabel(`amplitude (V)')
>> title(`A plot of voltage vs. time')
```

Describe what xlabel, ylabel, and title do in your lab report. Make sure to label your graphs in all your lab reports!

We can plot more than one signal on the same figure. Let's add a sinusoid at a lower frequency and amplitude. Use the hold command to add the new signal to the plot.

```
>> hold on
>> y = 0.5*cos(2*pi*349.23*t);
>> plot(t,y)
```

Using `hold on` allows us to plot a new signal without erasing the signals already on the plot. If you type `hold off`, the figure will no longer keep the existing signals when you type `plot`. Typing `hold` by itself toggles the state.

Instead of plotting these two signals on top of each other, we can plot them in separate windows in the same figure using the `subplot` command. Let's plot these two signals and their sum:

```
>> figure;
>> subplot(3,1,1)
>> plot(t,x)
>> xlabel(`time (s)'), ylabel(`voltage (V)')
>> subplot(3,1,2)
>> plot(t,y)
>> xlabel(`time (s)'), ylabel(`voltage (V)')
>> subplot(3,1,3)
>> plot(t,x+y)
>> xlabel(`time (s)'), ylabel(`voltage (V)')
>> title('Three signals in subplots')
```

The first two arguments to `subplot` specify the layout of windows within the figure. In our example, `subplot(3, 1, n)` specifies a three by one layout, so we have three subplots in a column. The third number, `n`, specifies which of the subplots to use. For rectangular layouts, `n` itemizes subplots by column (so `subplot(3,2,4)` would plot in the upper right of a three by two layout).

MATLAB offers a lot more plotting functionality. Step through the "Creating 2-D Plots" live demo to see some examples. (Type "demo" in MATLAB to display available demos in the help window if it is not still open.) Other useful plotting commands include `semilogx`, `semilogy`, `loglog`, and `axis`. Use `help` to learn about these.

The plot command connects each data point with a straight line, making a signal appear to be continuous-time. What plotting command from the demo could you use to emphasize that a signal is discrete-time? Explain your answer in your report.

Sounds

Note: MATLAB's ability to play sounds through speakers or headphones depends on how the host system is configured. If you are trying this on your own machine, you may need to change the settings.

Signals representing sound and imagery are extremely helpful for developing intuition. Let's use MATLAB to generate some sounds. A sampling rate of 44.1 kHz is typical for music. We will learn later in class that this allows recording of music with frequencies up to 22.05 kHz. Most humans cannot hear frequencies above about 20 kHz, so this sampling rate makes sense.

```
>> Fs = 44100; % 44.1 kHz sampling rate
>> Ts = 1/Fs; % this is the sampling period or sampling interval
>> t = -0.5:Ts:0.5;
>> x =  0.3*cos(2*pi*440*t);
```

At this point we have created one second of audio in the vector x. The next command will send this signal to the speakers at the appropriate sampling rate. Before you hit `Enter`, be sure to turn the volume down (especially on headphones). You can turn it up latter as necessary.

```
>> sound(x,Fs)
```

You should hear one second of a pure 'A' (440 Hz).

We can make more interesting sounds with the help of some new MATLAB commands. The commands `zeros(m, n)` and `ones(m, n)` are extremely useful. They create m-by-n arrays of zeros and ones, respectively:

```
>> zeros(2,5)

ans =

     0     0     0     0     0
     0     0     0     0     0

>> ones(1,3)

ans =

     1     1     1
```

We can create a unit step signal using `zeros(m, n)` and `ones(m, n)` as follows.

```
>> u = [zeros(1,22050), ones(1,22051)];
```

Notice the notation we used to create the vector `u`. In general, typing `[A, B]` will concatenate the matrices (or vectors) `A` and `B`. The comma (,) puts the matrices side-by-side, so `A` and `B` must have the same number of rows for this to work. The semicolon (;) concatenates matrices on top of one another (assuming they have the same number of columns).

Now lets make another sound signal using our unit step function.

```
>> y = 0.3*cos(2*pi*349.23*t);
>> s = x + u.*y;
```

The last line adds the vector `x` to another vector formed by multiplying `y` by a unit step. The "dot multiply" notation (.*) tells MATLAB to multiple the vectors point-wise rather than using a vector product. Now we are ready to play our new signal.

```
>> sound(s,Fs)
```

Describe what you hear in your lab report.

Images

Usually we deal with signals that are a function of one independent variable, such as the sound signals we just created. Signals can be functions of two or more independent variables as well. Images are two-dimensional signals where the two independent variables are the x and y coordinates. The dependent variable is either a grayscale value or a color (often represented by three dependent variables: R, G, and B). Future labs will explore images and image processing; this section will briefly introduce images in MATLAB.

We can use `zeros(m, n)`, `ones(m, n)`, and concatenation to make a very boring image.

```
>> im = [zeros(20,20), ones(20,20); ones(20,20), zeros(20,20)];
>> figure; imshow(im)
```

We could make the image a little more exciting by repeating a few times.

```
>> board = repmat(im,4,4);
>> figure; imshow(board)
```

The command `repmat(im,4,4)` repeats the matrix `im` in an four by four grid.

Without other arguments, `imshow` assumes images to be grayscale with zero corresponding to black and one corresponding to white. Color images can be displayed by providing a colormap.

```
>> load clown.mat
>> figure; imshow(X,map)
```

MATLAB also provides the commands `image` and `imagesc` to display images. These assume color images, but the default colormap is not always appropriate. Pixel values should range from 0 to 255 when using `image` (assuming a colormap of 256 values). When using `imagesc`, the pixel values are automatically scaled to fit the colormap.

Try the following commands and explain any differences that you observe between `imshow`, `image`, and `imagesc`.

```
>> figure; imshow(X,map)
>> figure; image(X)
>> colormap(map)
>> figure; imagesc(X)
>> colormap(map)
```

Report Checklist

Be sure the following are included in your report.

1. Section 4.1.1: Code to solve DC circuit and solution
2. Section 4.1.2: Solution for $v_C(t)$
3. Section 4.1.3: Solution for $v_C(t)$ for four $v_S(t)$ input signals
4. Section 4.2.1: Description of what happens when plotting `x` without `t`
5. Section 4.2.1: Description of `xlabel`, `ylabel`, and `title`
6. Section 4.2.1: Description of how to emphasize a discrete-time signal in a plot
7. Section 4.2.2: Description of the sound you hear
8. Section 4.2.3: Explanation of the differences you observe

9 MathWorks MATLAB Website. *MATLAB*, The MathWorks, Inc., https://www.mathworks.com/products/matlab.html

"MATLAB Installation Instructions." *Iowa State University Information Technology*, Iowa State University of Science and Technology, www.it.iastate.edu/services/software-students/matlab

Eaton, John W. GNU Octave Website. *GNU Octave*, John W. Eaton, https://www.gnu.org/software/octave/

Scilab Website. *Scilab*, ESI Group, https://www.scilab.org/

FreeMat Website. *FreeMat*, SourceForge, http://freemat.sourceforge.net/

---

1. Pun intended.↩
2. `ans` is really a variable in MATLAB which automatically takes on the value of the last output.↩

This page titled 1.1: MATLAB Basics is shared under a CC BY-NC 4.0 license and was authored, remixed, and/or curated by Andrew K. Bolstad & Julie A. Dickerson (Iowa State University Digital Press) via source content that was edited to the style and standards of the LibreTexts platform.

1.2: Tuning Forks

Learning Objectives

By the end of this lab, students will be able to:

1. Use the CyDAQ to record data
2. Write and use a MATLAB function
3. Measure the period of a signal
4. Determine the fundamental frequency of a measured signal

Overview

Tuning forks are physical systems that generate sinusoidal signals. When a tuning fork is exposed to vibration, it disturbs nearby air molecules, creating regions of higher-than-normal pressure (called compressions) and regions of lower-than-normal pressure (called rarefactions). A microphone converts these variations to an electrical signal which can be digitized with an analog-to-digital converter (ADC). The digital signal can be recorded, analyzed, processed, and replayed through a digital-to-analog converter (DAC) and speaker. This lab allows you to analyze sounds produced by tuning forks.

Pre-Lab Reading

Read the lab manual and the "Getting Started with the CyDAQ" document. Also read the Wikipedia on tuning forks (in particular to see how the frequency of a fork is calculated). This lab uses tuning forks with frequencies ranging from 128 Hz to 4096 Hz. Assuming that the stiffness of the tuning forks is the same, how does the length of a tuning fork change as the frequency increases? Enter your answer in the Canvas pre-lab and verify your hypothesis when you get to the lab.

Do not strike the tuning forks on the table.

Lab Exercises

Overview

In this lab, you will be introduced to the CyDAQ, a device created specifically for this course. The CyDAQ is capable of recording signals from a variety of sensors, and storing the data in a way that can be easily used in MATLAB. For more information, refer to the 'Getting Started with CyDAQ' document on Canvas.

Using a tone generator smartphone app, a pure sinusoidal tone can be generated and recorded using the microphone on the CyDAQ. In order to observe a decaying signal, a tuning fork can also be used. The CyDAQ will convert the generated sound to an electrical signal, which in turn is converted to a sequence of numbers stored in a digital file which can be displayed on computer screen as a sinusoidal curve. Characteristics of the sound wave, such as its period T and frequency f can be determined from this curve. Knowing the wave's period, its frequency f is easily computed using the formula:

$$f = 1/T$$

MATLAB Preliminaries

In this section, you will write an m-file function that takes a sound signal at a specific sampling rate and plots the FFT (Fast Fourier Transform) of the signal. You will need to import the data recorded from the CyDAQ into MATLAB. Your function should take the 'data' vector imported from the CyDAQ, and the sampling rate, F_s.

Introduction to Using Functions in MATLAB

Functions are m-files that can accept input arguments and return output arguments. The names of the m-file and of the function should be the same. Functions operate on variables within their own workspace, separate from the workspace you access at the MATLAB command prompt. The first line of a function m-file starts with the keyword `function`. It gives the function name and order of arguments. A simple function, called `compmag5` that computes five times the magnitude of a complex number of the form $x + jy$ is given below. In this case, there are two input arguments and one output argument.

```
function [z] = compmag5(x,y);
% % % % % % % % %
% this function computes the magnitude of the complex number
% x+jy and returns it in variable z.
% Inputs:
% x - real part of complex number
% y - imaginary part of complex number
%Outputs:
%z - magnitude of complex number times 5
%written by J.A. Dickerson, January, 2005
a=5;
% compute magnitude and multiply
z = a * sqrt(x^2+y^2);
```

The next several lines, up to the first blank or executable line, are comment lines that provide the help text. These lines are printed when you type

```
>>help compmag5
```

The first line of the help text is the H1 line, which MATLAB displays when you use the `lookfor` command or search a directory with MATLAB's Current Folder browser. The rest of the file is the executable MATLAB code defining the function. The variable `a` introduced in the body of the function, as well as the variables on the first line, `m`, `x` and `y`, are all local to the function; they are separate from any variables in the MATLAB workspace. This means that if you change the values of any variables with the same name while you are in the function, it does not change the value of the variable in your workspace.

If no output argument is supplied, the result is stored in `ans`. You can terminate any function with an `end` statement but, in most cases, this is optional. Functions return when the end of the function is reached. The function is used by calling the function within MATLAB or from a script. For example, the commands below can be used to call compmag5 and get a result of $m = 14.1421$.

```
>> x=2; y=2; m=compmag5(x,y);
```

Writing an FFTPlot Function

For this lab, you will need a function that plots the frequency spectrum of a recorded signal. This function can also be reused in future labs. Using the FFT function in MATLAB will allow you to do so. For more information, refer to the following link: https://www.mathworks.com/help/matlab/ref/fft.html. Be sure to plot the amplitude spectrum with amplitude on the y-axis and frequency on the x-axis. You function should take as input arguments the 'data' vector imported from the CyDAQ, and the sampling frequency (F_s) used when collecting the signal.

The following will help you get started:

```
function [] = FFTPlot(data, Fs)
% FFTPlot: convert data vector from time domain to frequency
% domain
%   data: imported `data' vector from the CyDAQ
%   Fs: sampling frequency specified on the CyDAQ



end
```

Labeling Plots in MATLAB

It is very important to label all plots and graphs submitted with your lab report. Recall how to use the functions `xlabel`, `ylabel`, and `title` from Lab 1. The axis command can be used to set the beginning and end values for the graph axes. For example, if the time should start at 2 s and end at 5 s and the recorded signal should range between 0 and 10 volts, use the command `axis([2, 5, 0 10]);`. Type `help axis` for more information.

Signal Collection

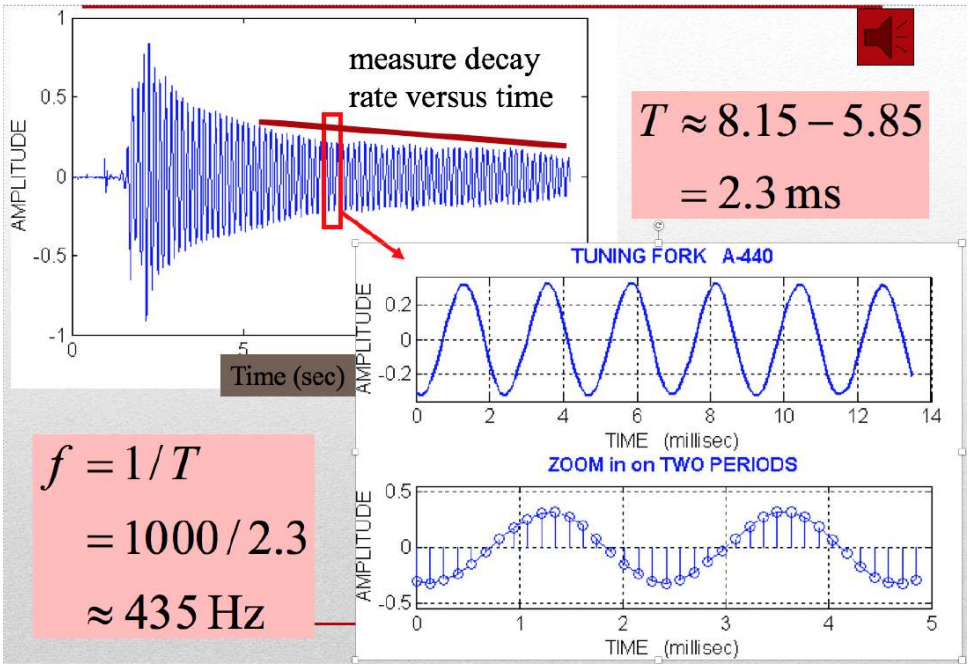
1. First, using the CyDAQ, collect signals generated by a tone generator app. Refer to the 'Getting Started with CyDAQ' document for instructions. Collect the signals using a sampling frequency of 8000 Hz on Channel 0, with 'No Filter' selected. Select two frequencies greater than 150 Hz and less than 1000 Hz. For each of the selected frequencies, plot both the time domain of the signal (`plot(time,data)`) and the frequency domain of the signal (`FFTPlot(data,Fs)`) in the same figure using subplot. Remember to include title and axis labels for both subplots! Include these plots in your report.
2. Use the data cursor tool to measure the frequency. In the frequency domain plot, this can be done by finding the x coordinate of a peak value. In the time domain plot, this can be done by measuring the time difference between two peaks. (The time difference between two consecutive peaks would give the period.) You may be able to achieve better accuracy by measuring the time it takes for N peaks to occur, then dividing by $N-1$. Verify that your two frequency measurements are reasonably close to each other. How do your measurements compare to the frequency you specified in the tone generator? Explain any discrepancies between your measured data and your expected value.
3. Next, collect tuning fork signals. Use the CyDAQ to collect signals for two different tuning forks with frequencies at or under 2048 Hz using a sampling frequency of 8000 samples/sec. Repeat parts A and B for each of the tuning forks. How do your measurements compare to the frequency listed on the tuning fork you used? Explain any discrepancies between your measured data and the expected frequency. Note: If your measured frequency is not reasonably close to the tuning fork frequency, it is possible that you are collecting data for too long. With your lab partner, have one person strike the tuning fork and one person push start and stop. Stop data collection within three seconds of starting.
4. 

Figure 1.2.1: Example tuning fork recording.
5. Comment on the other signals present in the spectrum. Why don't you see a nice clean delta function in the frequency plot?

6.

Signal Source	Frequency Estimate from Time Plot (Hz)	Magnitude Decay Rate (V/s)	Frequency Estimate from Frequency Plot (Hz)
Tuning fork labeled 1024			
Signal generator 440 Hz		N/A	

Report Checklist

Be sure the following are included in your report.

1. All your MATLAB code in an appendix of the report (including your FFTPlot function)
2. Section 4.3: Time and frequency plots of data measured from a tone generator app
3. Section 4.3: Time and frequency plots of data measured from a tuning fork
4. Section 4.3: Frequency measurements of the tone generator app's signal from time and frequency plots (report in table)
5. Section 4.3: Frequency measurements of the tuning fork from time and frequency plots (report in table)
6. Section 4.3: Explanation of discrepancies between measured and expected frequencies
7. Section 4.3: Plot and measurement of tuning fork amplitude decay (report estimate in table)
8. Section 4.3: Comment on other signals in the spectrum

This page titled 1.2: Tuning Forks is shared under a CC BY-NC 4.0 license and was authored, remixed, and/or curated by Andrew K. Bolstad & Julie A. Dickerson (Iowa State University Digital Press) via source content that was edited to the style and standards of the LibreTexts platform.

1.3: Impulse Response and Auralization

Learning Objectives

By the end of this lab, students will be able to:

1. Use a real-world impulse response to simulate different environments using discrete-time convolution
2. Describe FM modulated signals and chirp signals
3. Interpret a spectogram

Overview

"Auralization" is the process of using a computer to create or reproduce sound, often to achieve a desired sound effect. This is done by simulating characteristics of a room or other environment in order to produce sound that seems to have been recorded in that environment. Auralization is an increasingly popular technique for accurately simulating reverberation using acoustic impulse responses (IRs) from interesting buildings, spaces, and other sources. It is also used in video game and film production to create realistic sound effects.

Bring headphones to this lab!

Pre-Lab Reading

Auralization and Background

EchoThief (http://www.echothief.com) collects interesting IRs from "unusually clamorous places" for anyone to use. You can read about how these IRs are created here. A similar process is described in Section 3.1.1 This lab will use the computed impulse responses to approximate the sound of a signal in different environments.

Signal Acquisition

Ideally, we could obtain the acoustic impulse response of a space by producing an impulse signal and measuring the sound it produces in a given space. Unfortunately, impulse functions can only be approximated in the real world. One way to do this is to produce a short, tall (i.e., loud) pulse, but this requires a large amount of power (equal to the energy of the pulse divided by the pulse width). Requiring both more energy and a shorter pulse makes electrical power handling a limiting factor in such a system. The output stage of a transmitter can only handle so much power without destroying itself.

Chirp signals provide a way of breaking this limitation while still exciting a broad range of frequencies. (Knowing the impulse response is equivalent to knowing how the system responds to a sinusoid at any frequency.) A chirp signal is a sinusoid whose frequency changes from a starting frequency to an ending one. To generate a chirp, an impulse is passed through a chirp system. After the chirp echoes are received, the signal is passed through an antichirp system which converts the chirp echoes to an impulse response as shown in Figure 1 below. This phenomenon is called waveshaping. Often a logarithmically swept sine wave is used as a chirp signal.

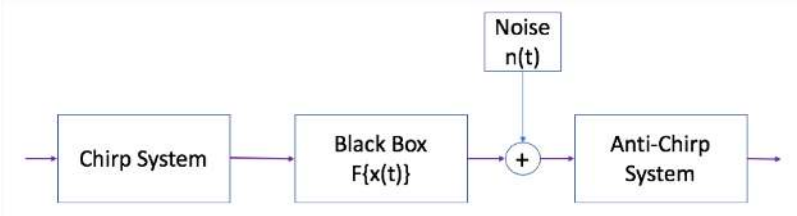


Figure 1.3.1: An impulse response measurement system

Chirp Signals

Many interesting signals can be produced by changing the argument of a generalized sinusoid as different functions of time. This is called *FM synthesis* or *Frequency Modulation*[1] and is used to create instrument simulations, interesting sound effects, improve

performance of radar systems, etc. A chirp signal is a sinusoid whose frequency sweeps from a starting frequency to an ending one. A "standard" sinusoid has

$$x(t) = A\cos\left(\Psi(t)\right) \qquad \Psi(t) = \omega t + \phi$$

However, much more interesting signals can be created with different functions such as a quadratic, logarithmic, or sinusoid as shown below.

$$\Psi(t) = 2\pi\mu t^2 + 2\pi f_0 t + \phi$$
$$\Psi(t) = e^{at} + 2\pi f_0 t + \phi$$
$$\Psi(t) = \cos\left(2\pi f_1 t\right) + 2\pi f_0 t + \phi$$

The frequency spectra of these signals are hard to analyze; however, a decent approximation of signal behavior can be found by looking at the first derivative of the argument function. This is known as the *instantaneous frequency*, f_i, of the signal. The instantaneous frequency of the linear, quadratic, and logarithmic chirp are:

$$\begin{aligned} Linear: &\quad f_i(t) = f_0 + \beta t &\quad \beta &= \frac{f_1 - f_0}{t_1} \\ Quadratic: &\quad f_i(t) = f_0 + \beta t^2 &\quad \beta &= \frac{f_1 - f_0}{t_1^2} \\ Logarithmic: &\quad f_i(t) = f_0\beta^t &\quad \beta &= \left(\frac{f_1}{f_0}\right)^{1/t_1} \end{aligned}$$

Figure 1.3.2 below shows the linear chirp system. For the problem of auralization, a logarithmically swept sinusoid is used.

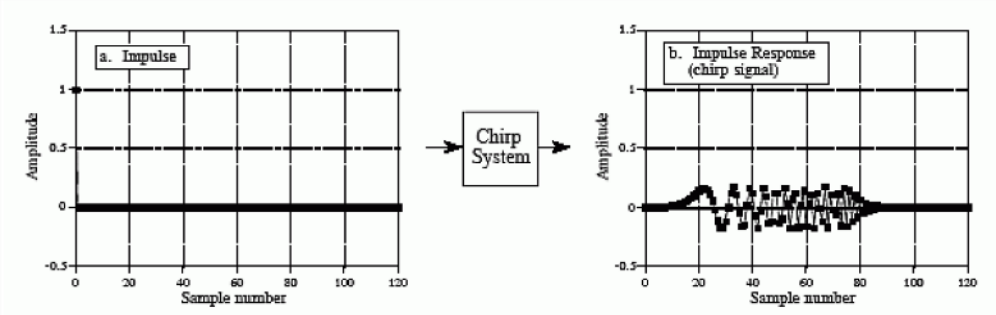


Figure 1.3.2: Impulse response of a linear chirp signal (from Figure 11-10 of The Scientist and Engineer's Guide to Signal Processing by Steven W. Smith,)

These signals can be generated using the Matlab command `chirp` .

```
y = chirp(t,f0,t1,f1,'method',phi)
```

The command generates samples of a swept-frequency cosine signal at the time instances defined in array `t` , where `f_0` is the instantaneous frequency at time 0, and `f_1` is the instantaneous frequency at time t_1. `f_0` and `f_1` are both in hertz. If unspecified, $f_0 = e^{-6}$ for logarithmic chirp and 0 for all other methods, $t_1 = 1$, and $f_1 = 100$.

Spectrograms

In addition to looking at signals in the time domain, it is often useful to look at the spectrum of a signal. A signal's spectrum shows which frequencies are present in the signal. A constant frequency sinusoid spectrum consists of two impulse functions at $\pm 2\pi f_0$. For more complicated signals, there may be many spikes. For even more complicated signals such as music or frequency modulated signals, the spectrum changes with time. In these cases, the spectrogram of a signal is used instead of a spectrum. A spectrogram is found by estimating the spectrum over multiple short windows of time. The magnitude of the spectrum over these time "windows" is plotted as intensity or color on a two dimensional plot with time on one axis and frequency on the other.

In Matlab, the function `spectrogram` will be used to compute the spectrogram. The spectrogram function divides a signal into segments. Longer segments provide better frequency resolution; shorter segments provide better time resolution. For more information, see http://www.mathworks.com/help/signal/examples/practical-introduction-to-time-frequency-analysis.html. There are theoretical limits on how well short pieces of the signal can represent the frequency content in a signal. Generally, longer time windows give better frequency resolution. The spectrogram function can be called as follows:

`spectrogram(xx, 1024, 512, [], Fs)` . The first argument, `xx` , is the time signal, 1024 is the number of samples in the time window, 512 is the number of samples of overlap between the time windows, and `Fs` is the sampling frequency. *NOTE: the spectrogram function requires the Matlab signal processing toolbox.*

In order to see what the spectrogram function does, run the following code.

```
N = 1024;
n = (0:N-1);
w0 = 2*pi/5;
x = sin(w0*n)+10*sin(2*w0*n);
spectrogram(x,128,64,[],100);
```

To see the spectrogram of a chirp, replace `x = sin(w0*n)+10*sin(2*w0*n);` with a chirp signal.

Lab Exercises

Using a Computed Impulse Response and Convolution

Implement sound effects on a provided sound clip and impulse responses of your choice.

1. Go to EchoThief and select an interesting location. Download the impulse response associated with the place you chose. The file you downloaded should be a *.wav* file.
2. Examine your impulse response. To do this, you will first need to load it into MATLAB using `[v, Fs] = audioread(file.wav)` . What is the sampling rate of this impulse response? How many samples long is your IR? How long in seconds is your IR? Plot your IR as a function of time, with the time in seconds. Comment on the plot in your report. Is this a mono or stereo recording? You can listen to your IR by typing `soundsc(v, Fs)` in MATLAB.
3. Download one of the two voice files from Canvas (female voice: *spfe49_1.wav* or male voice: *spme50_1.wav*). Load this audio file into MATLAB. What is the sampling rate of this file? Is this a mono or stereo recording? How many samples are in the signal? Listen to the audio. What is the voice talking about?
4. Now for the fun part! We want to make it sound like the voice is coming from the location corresponding to your IR. We can do this by convolving the speach signal with the impulse response. Suppose your voice signal is $v[n]$ and the IR is $h[n]$. The convolution $y[n] = v[n] * h[n]$ will produce a speach signal that sounds like it was recorded at the location you picked. It MATLAB, you can type `y = conv(v,h)` to convolve the signals.
    1. NOTE 1: make sure your voice signal and IR are recorded at the same sampling rate, otherwise you will need to use `resample` on one of them.
    2. NOTE 2: if your voice signal or IR (or both) were recorded in stereo, you will have to pick one of the two channels to use before calling `conv` . You can select the first column of a matrix $\mathbf{h}$ by typing `h1 = h(:,1)` .
5. You can play back your new voice by typing `soundsc(y,Fs)` .[2] Describe what you hear in your report.
6. Feel free to experiment with IRs from different locations and different sound files, but leave enough time to finish Section 4.2.

Investigating Frequency Modulated (FM) Signals

A chirp signal is also known as a swept frequency cosine. The frequency of the cosine changes in time. The frequency can change in many different ways as seen in Figure Figure 1.3.3 for linear and quadratic sweeps. We will focus on a linear sweep where the instantaneous frequency is linear as shown in Figure $1.3.3(left)$. Figure Figure $1.3.3(right)$ shows the case where the instantaneous frequency is a quadratic.

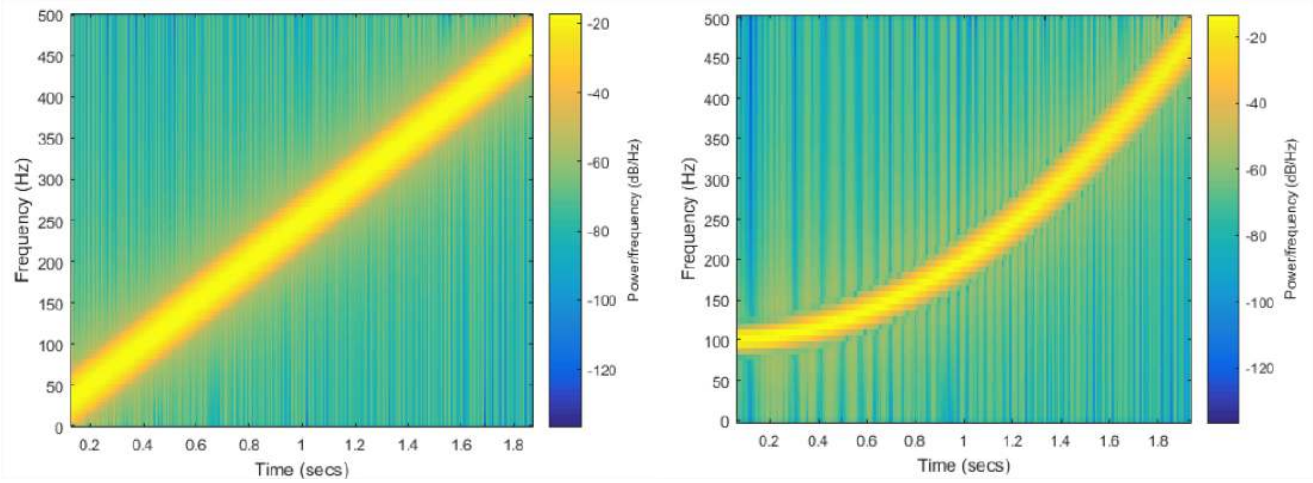


Figure 1.3.3: (left) Linear chirp and (right) quadratic chirp

1. Use the Matlab `chirp` command to generate a signal. Find the spectrogram using a window length of 2048 using the command:

```
spectrogram(x,2048,1024,[],Fs,`yaxis');
```

   Include this plot in your lab report and comment on the shape of the chirp.

2. What is the frequency value at $t = 1.0$ sec for the chirps in Figure 1.3.3 ?
3. As a test case, generate a chirp sound whose frequency starts at 400 Hz and ends at 4000 Hz with a duration of seven seconds and a sampling rate of 44.1 kHz. Listen to the sound that the chirp makes using the `soundsc` command in Matlab and include a listing of the commands that you used to generate the chirp signal and spectrogram.
4. Synthesize a “chirp” signal with the following parameters: time duration of five seconds with a sampling frequency of 11,025 Hz, starting frequency of 5000 Hz, and ending frequency of 20 Hz. Listen to the signal; what does it sound like (e.g., is the frequency movement linear)? Does it chirp down or up? Create a spectrogram of the signal to verify that you have the correct instantaneous frequency.
5. Synthesize a “chirp” signal with the following parameters: time duration of ten seconds with a sampling frequency of 11,025 Hz, starting frequency of 20 Hz, and the frequency at time $t = 1$ sec should be 1200 Hz. Listen to the signal; what does it sound like (e.g., is the frequency movement linear)? Does it chirp down or up? Create a spectrogram of the signal to verify that you have the correct instantaneous frequency. Speculate on what is happening with the signal.

Report Checklist

Be sure the following are included in your report.

Section 4.1

1. Characteristics of the impulse response you chose (length in sec, length in samples, sampling rate)
2. Plot of your IR and comments about it
3. Description of the voice you chose (sampling rate, mono/stero, number of samples, etc.)
4. Description of your sound effect

Section 4.2

1. Plot of your chirp with comments
2. Frequency values from Figure 1.3.3
3. Spectrogram of seven second chirp with code in the appendix
4. Description of five second chirp with spectrogram
5. Description of ten second chirp with spectrogram

9 “EchoThief Impulse Response Library,” *EchoThief*, SuperHoax, http://www.echothief.com/

Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical Publishing, San Diego, CA, 1997.

Michael Vorländer. *Auralization : Fundamentals of Acoustics, Modelling, Simulation, Algorithms and Acoustic Virtual Reality*, Springer, Berlin, Germany, 2008.

---

1. In FM radio, the argument of the sinusoid contains the audio signal.↩
2. Be sure to use `soundsc` . The `sound` function expects to get data ranging from -1.0 to 1.0 (see the help documentation), so if your audio signal exceeds 1.0 in magnitude, you will here additional distortion not caused by your IR.↩

This page titled 1.3: Impulse Response and Auralization is shared under a CC BY-NC 4.0 license and was authored, remixed, and/or curated by Andrew K. Bolstad & Julie A. Dickerson (Iowa State University Digital Press) via source content that was edited to the style and standards of the LibreTexts platform.

1.4: Frequency Response - Notch and Bandpass Filters

Learning Objectives

By the end of this lab, students will be able to:

1. Plot the frequency response of an FIR filter
2. Implement and apply an FIR filter in MATLAB
3. Design an FIR filter for nulling frequency components
4. Design an FIR filter for isolating specific frequencies

Overview

The goal of this lab is to study the response of finitie impulse response (FIR) filters to inputs such as complex exponentials and sinusoids. In the experiments, you will use MATLAB's `conv` function to implement filters in the time domain and `freqz` to obtain each filter's frequency response. As a result, you should learn how to characterize a filter by knowing how it reacts to different frequency components in the input. This lab also introduces two practical filters: bandpass filters and nulling filters. Bandpass filters can be used to detect and extract information from sinusoidal signals, e.g., tones in a touch-tone telephone dialer. Nulling filters can be used to remove sinusoidal interference, e.g., jamming signals in a radar or 60 Hz power signals from lights.

Pre-Lab Reading

Frequency Response of FIR Filters

The output or *response* of a filter for a complex sinusoidal input, $e^{j\Omega}$, depends on the frequency, Ω. Often a filter is described solely by how it affects different input frequencies–this is called the frequency response. For example, the frequency response of the two-point averaging filter

$$y[n] = \frac{1}{2}x[n] + \frac{1}{2}x[n-1]$$

can be found by using a general complex exponential as an input and observing the output or response.

$$\begin{aligned} x[n] &= Ae^{j(\Omega n+\phi)} \\ y[n] &= \frac{1}{2}Ae^{j(\Omega n+\phi)} + \frac{1}{2}Ae^{j(\Omega(n-1)+\phi)} \\ &= Ae^{j(\Omega n+\phi)}\frac{1}{2}\left(1+e^{-j\Omega}\right) \\ &= Ae^{j(\Omega n+\phi)}H\left(e^{j\Omega}\right) \end{aligned} \tag{1.4.1}$$

In [eq:freqresp] there are two terms, the original input and a term that is a function of Ω. This second term is the frequency response, and it is commonly denoted by $H(e^{j\Omega})$, which in this case is

$$H(e^{j\Omega}) = \frac{1}{2}\left(1+e^{-j\Omega}\right)$$

Once the frequency response, $H(e^{j\Omega})$, has been determined, the effect of the filter on any complex exponential may be determined by evaluating $H(e^{j\Omega})$ at the corresponding frequency. The output signal, $y[n]$, will be the original input complex exponential signal times a complex number. The phase of this number imparts a phase shift and the magnitude describes the gain applied to the complex exponential. The frequency response of a general finite impulse response (FIR) linear, time-invariant system is

$$H(e^{j\Omega}) = \sum_{k=0}^{M} b_k e^{-j\Omega k} \tag{1.4.2}$$

In the example above, $M = 1$ and $b_0 = b_1 = \frac{1}{2}$.

MATLAB Function for Frequency Response

MATLAB has a built-in function called `freqz` for computing the frequency response of a discrete-time LTI system. The following MATLAB statements show how to use `freqz` [1] to compute and plot both the magnitude (absolute value) and the phase of the frequency response of a two-point averaging system as a function of Ω in the range $-\pi \leq \Omega \leq \pi$:

```
bb = [0.5, 0.5]; \% Filter Coefficients
ww = -pi:(pi/100):pi; \% omega
HH = freqz(bb, 1, ww);
subplot(2,1,1);
plot(ww, abs(HH));axis([-pi,pi,0,1]);
subplot(2,1,2);
plot(ww, angle(HH));axis([-pi,pi,-2,2]);
xlabel(`Normalized Radian Frequency')
```

For FIR filters, the second argument of `freqz` must always be equal to 1. The frequency vector `ww` should cover an interval of length 2π for Ω, and its spacing must be fine enough to give a smooth curve for $H(e^{j\Omega})$. Note: we will always use capital `HH` for the frequency response in this lab.

Periodicity of the Frequency Response

The frequency responses of discrete-time filters are always periodic with period equal to 2π. Explain why this is the case by stating a definition of the frequency response and then considering two input sinusoids whose frequencies are Ω and $\Omega + 2\pi$:

$$x_1[n] = e^{j\Omega n}$$
$$x_2[n] = e^{j\Omega n + 2\pi n}$$

Notice that $x_2[n] = x_1[n]$, so the output will be the same for both inputs. Include your answer in your lab report.

The implication of periodicity is that a plot of $H(e^{j\Omega})$ only needs to extend over the interval $-\pi \leq n \leq \pi$ or any other interval of length 2π.

Lab Exercises

Frequency Response of the Four-Point Averager

Filters that average input samples over a certain interval are called “running average” filters or “averagers,” and they have the following form (for an L-point averager):

$$y[n] = \frac{1}{L}\sum_{k=0}^{L-1} x[n-k]$$

In other words, the impulse response of an L-point averager is:

$$h[n] = \begin{cases} \frac{1}{L} & 0 \leq n \leq L-1 \\ 0 & \text{otherwise} \end{cases}$$

1. Use Euler’s formula, [eq:frFIR], and complex number manipulations to show that the frequency response for the 4-point running average operator is given by:

$$H\left(e^{j\Omega}\right) = \frac{\cos\left(0.5\Omega\right) + \cos\left(1.5\Omega\right)}{2} e^{-j1.5\Omega} \tag{1.4.3}$$

2. Implement [eq:4pnt] directly in MATLAB and plot the frequency response. Use a vector that includes 400 samples between $-\pi$ and π for Ω. Since the frequency response is a complex-valued quantity, use `abs` and `angle` to extract the magnitude and phase of the frequency response for plotting. Plotting the real and imaginary parts of $H(e^{j\Omega})$ is not very informative.
3. Use `freqz` in MATLAB to compute $H(e^{j\Omega})$ numerically (from the filter coefficients) and plot its magnitude and phase versus Ω. Include the appropriate MATLAB code to plot both the magnitude and phase of $H(e^{j\Omega})$ in your lab report’s appendix. Follow the example in Section 3.2. The filter coefficient vector for the 4-point averager is defined via:

```
bb = 1/4*ones(1,4);
```

Note: the function `freqz(bb,1,ww)` evaluates the frequency response for all frequencies in the vector `ww`. It uses the summation in [eq:frFIR], not the formula in [eq:4pnt]. The filter coefficients are defined in the assignment to vector `bb`. How do your results compare with part (b)?

The MATLAB `find` Function

Often signal processing functions are performed in order to extract information that can be used to make a decision. The decision process inevitably requires logical tests, which might be done with `if-then` constructs in MATLAB. However, MATLAB permits vectorization of such tests, and the `find` function is one way to do lots of tests at once. In the following example, `find` extracts all the numbers that "round" to 3:

```
xx = 1.4:0.33:5, jkl = find(round(xx)==3), xx(jkl)
```

The argument of the `find` function can be any logical expression. Notice that `find` returns a list of indices where the logical condition is true. See `help` on `relop` for information.

Now, suppose that you have a frequency response:

```
ww = -pi:(pi/500):pi; HH = freqz( 1/4*ones(1,4), 1, ww );
```

Use the `find` command to determine the indices where `HH` is zero, and then use those indices to display the list of frequencies where `HH` is zero. Since there might be round-off error in calculating `HH`, the logical test should probably be a test for those indices where the magnitude (absolute value in MATLAB) of `HH` is less than some rather small number, e.g., 1×10^{-6}. Compare your answer to the frequency response that you plotted for the four-point averager in Section 4.1.

Nulling Filters for Rejection

Nulling filters are filters that completely eliminate some frequency component. If the frequency is $\Omega = 0$ or $\Omega = \pi$, then a two-point FIR filter will do the nulling. The simplest possible general nulling filter can have as few as three coefficients. If Ω is the desired nulling frequency, then the following length-3 FIR filter will have a zero in its frequency response at $\Omega = \Omega_n$:

$$y[n] = x[n] - 2\cos(\Omega)x[n-1] + x[n-2]$$

For example, a filter designed to completely eliminate signals of the form $Ae^{j0.5\pi n}$ would have the following coefficients because we would pick the desired nulling frequency to be $\Omega_n = 0.5\pi$.

$$b_0 = 1, \qquad b_1 = -2\cos(0.5\pi) = 0, \qquad b_2 = 1$$

1. Design a filtering system that consists of the cascade of two FIR nulling filters that will eliminate the following input frequencies: $\Omega = 0.44\pi$, and $\Omega = 0.7\pi$. For this part, derive the filter coefficients of both nulling filters.
2. Generate an input signal $x[n]$ that is the sum of three sinusoids:

   $$x[n] = 5\cos(0.3\pi n) + 22\cos\left(0.44\pi n - \frac{\pi}{3}\right) + 22\cos\left(0.7\pi n - \frac{\pi}{4}\right)$$

   Make the input signal 150 samples long over the range $0 \leq n \leq 149$.
3. Use `conv` to filter the sum of three sinusoids signal $x[n]$ through the filters designed in part (a). Include the MATLAB code that you wrote to implement the cascade of two FIR filters in the appendix.
4. Make a plot of the output signal and show the first 40 points. Determine (by hand) the exact mathematical formula (magnitude, phase, and frequency) for the output signal for $n \geq 5$. (Hint: recall that an LTI system alters the magnitude and phase of sinusoidal inputs.)

Simple Bandpass Filter Design

The L-point averaging filter is a lowpass filter. Its passband width is controlled by L, being inversely proportional to L. It is also possible to create a filter whose passband is centered around some frequency other than zero. One simple way to do this is to define the impulse response of an L-point FIR as:

$$h[n] = \frac{2}{L}\cos(\Omega_c n), \qquad 0 \le n < L$$

where L is the filter length and Ω_c is the center frequency that defines the frequency location of the passband. For example, we would pick $\Omega_c = 0.44\pi$ if we want the peak of the filter's passband to be centered at 0.44π. The bandwidth of the bandpass filter (BPF) is controlled by L; the larger the value of L, the narrower the bandwidth.

1. Generate a bandpass filter that will pass a frequency component at $\Omega = 0.44\pi$. Make the filter length (L) equal to 10. Since we are going to be filtering the signal defined below, measure the gain of the filter at the three frequencies of interest: $\Omega = 0.3\pi$, $\Omega = 0.44\pi$ and $\Omega = 0.7\pi$.

$$x[n] = 5\cos(0.3\pi n) + 22\cos\left(0.44\pi n - \frac{\pi}{3}\right) + 22\cos\left(0.7\pi n - \frac{\pi}{4}\right) \tag{1.4.4}$$

2. The passband of the BPF is defined by the region of the frequency response where $|H(e^{j\Omega})|$ is close to its maximum value. If we define the maximum to be H_{max}, then the passband width is defined as the length of the frequency region where the ratio $|H(e^{j\Omega})|/H_{max}$ is greater than $1/\sqrt{2} = 0.707$. The stopband of the BPF filter is defined by the region of the frequency response where $|H(e^{j\Omega})|$ is close to zero. In this case, we will define the stopband as the region where $|H(e^{j\Omega})|$ is less than 25% of the maximum.

   Figure 1 shows how to define the passband and stopband. Note: you can use MATLAB's `find` function to locate those frequencies where the magnitude satisfies $|H(e^{j\Omega})| \ge 0.707 H_{max}$.

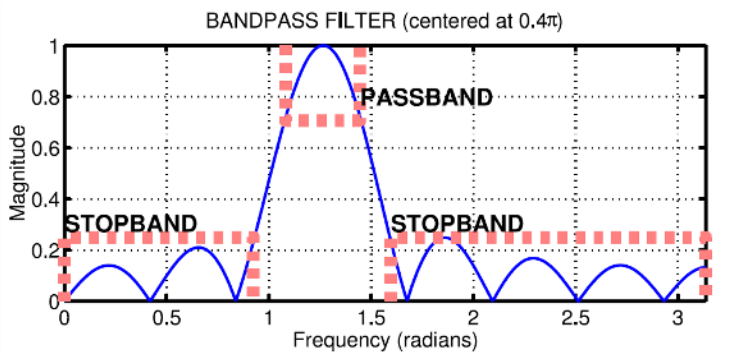


Figure 1.4.1: Passband and Stopband for a typical FIR bandpass filter. In this case, the maximum value is 1, the passband is the region where the frequency response is greater than $1/\sqrt{2} = 0.707$ times the maximum value, and the stopband is defined as the region where the frequency response is less than 25% of the maximum.

   Make a plot of the frequency response for the $L = 10$ bandpass filter from part (a), and determine the passband width (at the 0.707 level). Repeat the plot for $L = 20$ and $L = 40$ so you can explain how the width of the passband is related to filter length L, i.e., what happens when L is doubled or halved.

3. Comment on the selectivity of the $L = 10$ bandpass filter. In other words, which frequencies are "passed by the filter"? Use the frequency response to explain how the filter can pass one component at $\Omega = 0.44\pi$, while reducing or rejecting the others at $\Omega = 0.3\pi$ and $\Omega = 0.7\pi$.
4. Generate a bandpass filter that will pass the frequency component at $\Omega = 0.44\pi$, but now make the filter length (L) long enough so that it will also greatly reduce frequency components at (or near) $\Omega = 0.3\pi$ and $\Omega = 0.7\pi$. Determine the smallest value of L so that
   1. Any frequency component satisfying $|\Omega| \le 0.3\pi$ will be reduced by a factor of 10 or more[2].
   2. Any frequency component satisfying $0.7\pi \le |\Omega| \le \pi$ will be reduced by a factor of 10 or more.

   This can be done by making the passband width very small.

5. Use the filter from the previous part to filter the "sum of three sinusoids" signal in 1.4.4. Make a plot of 100 points of the input and output signals, and explain how the filter has reduced or removed two of the three sinusoidal components.
6. Make a plot of the frequency response (magnitude only) for the filter from part (d), and explain how $H(e^{j\Omega})$ can be used to determine the relative size of each sinusoidal component in the output signal. In other words, connect a mathematical description of the output signal to the values that can be obtained from the frequency response plot.

Report Checklist

Be sure the following are included in your report.

1. Section 3.3: explanation of periodicity of frequency response
2. Section 4.1: derivation of [eq:4pnt]
3. Section 4.1: (labeled) plots of magnitude and phase of $H(e^{j\Omega})$
4. Section 4.1: plots of $H(e^{j\Omega})$ using `freqz` (code in appendix)
5. Section 4.2: list of frequencies where `HH` is 0 (and compare to above)
6. Section 4.3: coefficients of two nulling filters
7. Section 4.3: MATLAB code for filtering signal (appendix)
8. Section 4.3: plot of nulled output and formula for output signal
9. Section 4.4: coefficients of bandpass filter; gain at three frequencies
10. Section 4.4: plots of frequency response for $L = 10$, 20, and 40 with a description of the passband widths
11. Section 4.4: comments on selectivity of the $L = 10$ BPF
12. Section 4.4: the smallest L that satisfies the criteria
13. Section 4.4: plot of input and output; explanation of effect on three sinusoid signal
14. Section 4.4: plot of frequency response; explanation of size of each sinusoidal component in output

---

1. If the output of the `freqz` function is not assigned, then plots are generated automatically; however, the magnitude is given in decibels which is a logarithmic scale. For linear magnitude plots a separate call to `plot` is necessary.↩
2. For example, the input amplitude of the 0.7π component is 22, so its output amplitude must be less than 2.2↩

This page titled 1.4: Frequency Response - Notch and Bandpass Filters is shared under a CC BY-NC 4.0 license and was authored, remixed, and/or curated by Andrew K. Bolstad & Julie A. Dickerson (Iowa State University Digital Press) via source content that was edited to the style and standards of the LibreTexts platform.

1.5: Fourier Series

Learning Objectives

By the end of this lab, students will be able to:

1. Use MATLAB to approximate Fourier series coefficients of a sampled signal.
2. Use MATLAB to synthesize a signal from its Fourier series coefficients.
3. Describe the difference in sound made by two musical instruments in terms of their Fourier series coefficients.
4. Create your own synthesized instrument using Fourier series (bonus).

Overview

In this lab you will experience the power of Fourier series analysis and synthesis.

Pre-Lab Reading

Fourier Analysis

Begin by reviewing the Fourier analysis formula. Given a periodic signal $x(t)$ with period T, $x(t)$ can be represented by

$$x(t) = \sum_{k=-\infty}^{\infty} a_k e^{jk\omega_0 t}$$

where $\omega_0 = \frac{2\pi}{T}$ and

$$a_k = \frac{1}{T}\int_0^T x(t)e^{-jk\omega_0 t}dt \qquad \text{for all } k$$

Although an infinite number of harmonics may be required for a general signal, in most situations, a finite number of them provides a practically good approximation.

If the signal $x(t)$ is not given as a mathematical function but we have a sampled recorded version of it, the integrals to compute the coefficients (a_k) cannot be evaluated precisely. Although there are more efficient methods to perform the Fourier analysis directly on the sampled signals (e.g., the Fast Fourier Transform (FFT)), we will use a simple method to approximately evaluate those integrals: the Riemann approximation. Assuming that the signal $x(t)$ is reasonably smooth (Riemann integrable) and that the sampling time T_s is an integer fraction of the period T, i.e., $T = NT_S$ of samples in a period (this is not really necessary but simplifies our code), then:

$$a_k = \frac{1}{T}\int_0^T x(t)e^{-jk\omega_0 t}dt \simeq \frac{T_s}{T}\sum_{n=0}^{T/T_s} x[T_s n]e^{-jk\omega_0 T_s n}. \tag{1.5.1}$$

The approximation gets better and better as the number of samples goes to infinity (meaning T_s goes to zero). In other words, fixing the sampling time limits the quality of the approximation, especially for larger values of k. This is because T_s becomes too large with respect to $k\omega_0$, the frequency of the kth harmonic.

Find the Fourier series coefficients a_0, a_1, a_2, and a_3 for the square wave given below (assume the signal repeats with period $T = 0.04$). Use the integral in [eq1.eq] and include your answer in your report.

$$x(t) = \begin{cases} 1 & 0 \le t < 0.02 \\ 0 & 0.02 \le t < 0.04 \end{cases}$$

Fourier Synthesis

The synthesis formula allows one to generate periodic signals from the linear combination of harmonic complex exponentials. Here we assume that the number of non-zero coefficients a_k is finite. Then

$$\tilde{x}(t) = \sum_k a_k e^{jk\omega_0 t}$$

In particular,

$$\tilde{x}[T_s n] = \sum_k a_k e^{jk\omega_0 T_s n}$$

Lab Exercises

Fourier Series Analysis Function

Write a Matlab function “findFS" which takes as inputs x_T, a vector of samples of $x(t)$ covering one period of $x(t)$; k, the index of the desired coefficient; the period T; and the sampling time T_s; and produces as output the approximate a_k coefficient according to (1.5.1). Note that you can easily take advantage of vectorization by considering the product of the row vector x_T and the column vector $e^{-jk\omega_0 T_s n}$. A template for the function is provided.

Fourier Series Coefficients of a Square Wave

Let’s construct a test signal. Define in Matlab

```
x_T=[ones(1000,1);zeros(1000,1)];
```

This is one period of a square wave signal. Let the fundamental period be $T = 0.04$ sec.

- Calculate the sampling time T_s and explain your answer. (Hint: $T_s = T/2000$)
- Use the function you wrote in the previous exercise to compute a_0, a_1, a_2, and a_3.
- Find a_{-1}, a_{-2}, and a_{-3}. How do these relate to a_1, a_2, and a_3?
- Verify the coefficients you have computed approximate reasonably well the actual coefficients you obtained for the prelab.

Fourier Series Synthesis

Write a Matlab function “synthFS” which has the following inputs: a column vector $C = [a_0; a_1; \ldots; a_m]$ of coefficients where m is the largest integer corresponding to a non-zero coefficient, the sampling time T_s, and the fundamental period T. The output should be $\tilde{x}_T(T_s n)$, the vector of $N = T/T_s$ (N an integer) samples corresponding to one period of $\tilde{x}$. Note that your program should verify that T/T_s is an integer. You will also need to extend C to include the complex conjugate coefficients corresponding to the negative k s. Finally, you may want to generate an array `F` whose columns are the vectors $e^{jk\omega_0 T_s n}$. $\tilde{x}_T$ is then computed as `F*CC` , where `CC` is a vector containing all the coefficients from $-m$ to m. A template for this function is provided.

Spectra of a Trumpet and Whistle

Load the data file `trumpet_whistle.mat` , which can be found on Canvas. It contains vectors `trumpet` and `whistle` . Both signals are sampled at $f_0 = 44100$ Hz, and they represent one period of synthetic trumpet and whistle tones generated by a toy electric keyboard.

- Compute the period of the two signals.
- For each signal, compute and report the first 9 harmonic coefficients, i.e., a_ks with $k = -9, \ldots, 9$ using the function “findFS” you have developed.
- For each signal, plot the magnitude and phase spectra of the frequency components from part b. Briefly comment on their main differences. You may use the Matlab function `stem` to plot the spectrum.
- For each signal, synthesize an approximation by using the periods in part a, the coefficients in part b, the sampling time $T_s = 1/f_0$, and your function “synthFS.”
- For each signal, plot on the same plot the given signal and its synthesized approximation. (You may use `spectraplot.m` found on Canvas. Make sure the axis labels are accurate.) Comment on the quality of the approximation in both cases.
- For each signal and its approximation, generate a new signal by repeating them for 1000 periods (e.g., using the built in function `repmat`). Use the function `sound` in Matlab and the sampling frequency f_0 to hear the sound you have generated. Can you hear the difference between the originals and the approximations? Comment. (Note you may need to scale the signals to have magnitude 1, see `help sound` .)

Note that the trumpet sound has a much richer spectrum and corresponds to a more complex sound. While the whistle sound essentially does not have any harmonic components above the ninth and does not have any even harmonic components, the trumpet sound has important harmonics above the ninth; this can be argued from the error in the approximation.

Bonus

Synthesize your own “instrument” by repeating steps d-f of the last subsection. Instead of using the a_k that you calculated for the trumpet and whistle, come up with your own coefficients any way you want. (You don’t have to limit yourself to nine coefficients.) Upload your sound and describe it in your report.

Report Checklist

Be sure the following are included in your report.

1. Section 4.1: include your function in the appendix
2. Section 4.2: provide answers for a-d
3. Section 4.3: include your function in the appendix
4. Section 4.4: provide answers for a-c, e, and f
5. Section 4.5: (bonus) describe the sound you created and explain how you created it

---

This page titled 1.5: Fourier Series is shared under a CC BY-NC 4.0 license and was authored, remixed, and/or curated by Andrew K. Bolstad & Julie A. Dickerson (Iowa State University Digital Press) via source content that was edited to the style and standards of the LibreTexts platform.

1.6: Introduction to Digital Images

Learning Objectives

By the end of this lab, students will be able to:

1. Describe digital image types, including monochrome and color.
2. Display images in MATLAB.
3. Perform basic pixel manipulation including clearing, copying, inverting, and thresholding.
4. Perform grsyscale brightening and contrast-stretching.

Overview

In this lab we introduce digital images as a new higher dimensional signal type. Digital images are written as two-dimensional matrices of numbers that can be manipulated to enhance contrast, invert images, and highlight objects.

Pre-Lab Reading

Monochrome Images

An image can be represented as a function $J(x, y)$ of two continuous variables representing the horizontal (x) and vertical (y) coordinates of a point in space. The variables x and y represent spatial dimensions. Thus, their units would be inches or some other unit of length. Moving images (such as video) add a time variable to the two spatial variables. Digital images sample the signal $J(x, y)$ at uniform intervals:

$$J[m, n] = J(mT_1, nT_2) \qquad 1 \leq m \leq M \text{ and } 1 \leq n \leq N$$

where T_1 and T_2 are the sample spacings in the horizontal and vertical directions. In MATLAB, we can represent an image as a matrix which consists of M rows and N columns. The matrix entry at (m, n) is the sample value $J[m, n]$—called a *pixel* (short for picture element).

Monochrome images are displayed using black and white and shades of gray, so they are called grayscale images. In this lab, we will consider only sampled grayscale still images. A sampled grayscale still image would be represented as a two-dimensional array of numbers. An important property of light images such as photographs and TV pictures is that their values are always non-negative and finite in magnitude; i.e.,

$$0 \leq J[m, n] \leq J_{max} < \infty$$

This is because light images are formed by measuring the intensity of reflected or emitted light which must always be a positive, finite quantity. When stored in a computer or displayed on a monitor, the values of $J[m, n]$ have to be scaled relative to a maximum value J_{max} . Usually, an eight-bit integer representation is used. With 8-bit integers, the maximum value (in the computer) would be $J_{max} = 2^8 - 1 = 255$, and there would be $2^8 = 256$ different gray levels for the display, from 0 to 255. Images in this format have type uint8 (unsigned 8-bit integer) in MATLAB. However, in image recognition applications, binary images ($J_{max} = 1$) are often used to separate parts of the image that are of interest from areas that are not. A simple example is the application of zip code recognition on letters. The algorithm would first search for the zip code, then filter out the rest of the image.

Displaying and Exporting Images in MATLAB

Most of the lab exercises in this class will require you to produce one or more output images. These will need to be incorporated into your report. Since many exercises will use MATLAB, we will describe here some basic I/O and display commands in the MATLAB environment.

- Reading Images: You can read an image file, *img.tif* or *image.jpg*, into the MATLAB workspace using the command:

```
IMG=imread('pout.tif');
```

  This will produce an image matrix IMG of data type uint8.

- Displaying Images: You can display the image array `IMG` with the following commands.

```
colormap(gray(256));  % only needed for grayscale image
imshow(IMG)
truesize;
```

If `IMG` is a grayscale image, the colormap function is needed to tell MATLAB which color to display for each possible pixel value. The `truesize` command, which is included in the image processing toolbox, maps each image pixel to a single display pixel to avoid any interpolation on the display. This will be important in future labs.

- Writing Images: When producing a lab report document, you should strive to present the best representation of your output images. Therefore, it is best to export your images to a lossless file format, such as TIFF or BMP, which can then be imported into your lab report document. You can write the image array `X` to a file using the `imwrite` function:

```
imwrite(x,'img_out.tif')
```

Note that if `X` is of type uint8, the `imwrite` function assumes a dynamic range of $[0, 255]$, and will clip any values outside that range. If `X` is of type double, then `imwrite` assumes a dynamic range of $[0, 1]$, and will linearly scale to the range $[0, 255]$, clipping values outside that range, before writing out the image to a file. To convert the image to type uint8 before writing, you can use

```
imwrite(uint8(x),'img_out.tif')
```

If your image is in color, such as RGB (Red-Green-Blue), then you will need to convert it to grayscale using the `rgb2gray` command in MATLAB:

```
RGB = imread(`peppers.png'); imshow(RGB)
I = rgb2gray(RGB);
figure; imshow(I)
```

Prelab Questions

Open Matlab and run the following command:

```
moon = imread(`moon.tif');
```

1. What is the size of the image example? NxM pixels
2. What is the variable type?
3. Write out one line of Matlab code for seeing the intensity value of any given pixel in the image.

Image Manipulation

The goal of image manipulation is to improve the image in ways that increase the performance of a system for human viewing or computer vision applications.

Histogram Computation

Image histograms are a simple but very useful method to analyze images and to enhance the quality of the image (at least for a human observer) by performing point transformations. In grayscale images, the histogram looks at the distribution of the intensity values of the image. For each range of grayscale values, the number of pixels in that range are counted. To see the distribution of intensities in the image, create a histogram by calling the `imhist` function. Generally, a good contrast image will have values spread out across the range of the intensity values from 0 to N. Figure $1.6.1$ shows the MATLAB image tire.tif and its histogram. Note that most of the pixels have low values that are in the range of 0-25 in intensity.

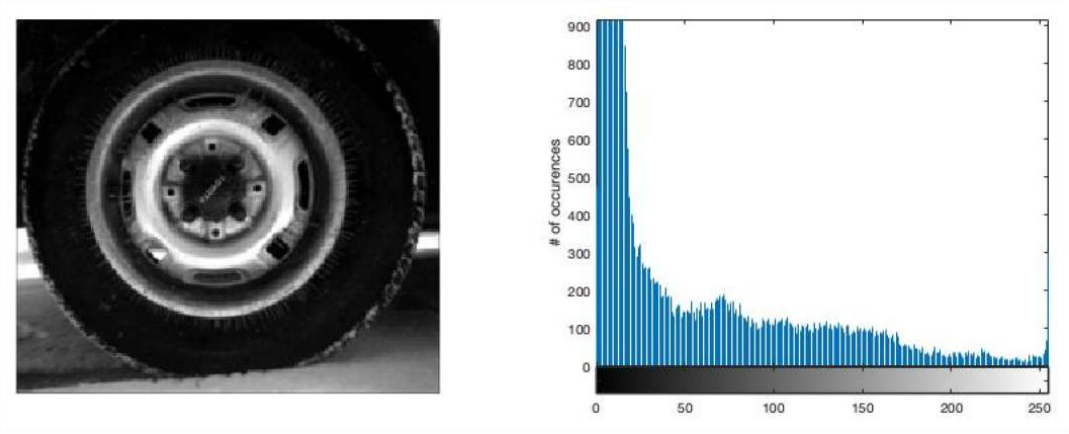


Figure 1.6.1: Image of a tire with a histogram of intensity values. Most of the pixels have low intensities.

Contrast Stretching and Intensity Level Slicing

Low contrast images result from poor illumination or lack of response from the imaging sensor. The idea behind contrast stretching is to increase the dynamic range of the gray levels in the image being processed. This is implemented by a mapping function, $s = T(r)$. Figure 2 shows a typical transformation for contrast stretching. The locations of points $(r1, s1)$ and $(r2, s2)$ control the shape of the contrast function. If $r1 = r2$ and $s1 = s2$, then the transformation is a linear function. If $r1 = r2$, $s1 = 0$, and $s2 = L - 1$, then the transformation is a thresholding function. Intermediate values produce different types of spread. The mapping is a piecewise linear interpolation.

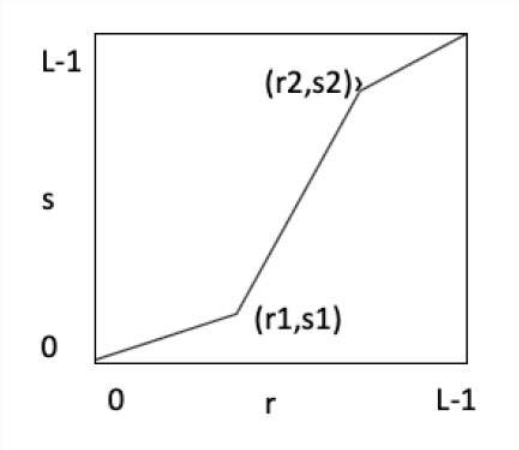


Figure 1.6.2: Example contrast mapping $s = T(r)$. The function T is applied to each pixel.

A related transformation is Intensity Level Slicing which can highlight or delete a specific range of gray levels. Applications include enhancing features such as water in satellite images or the tire in the image above. There are two main approaches for this: display a high value for all of the pixels in a desired range and zero out the others or brighten the pixels in the range and leave the other pixels the same. Levels can be deleted using a similar approach. Examples are shown in Figure 1.6.3.

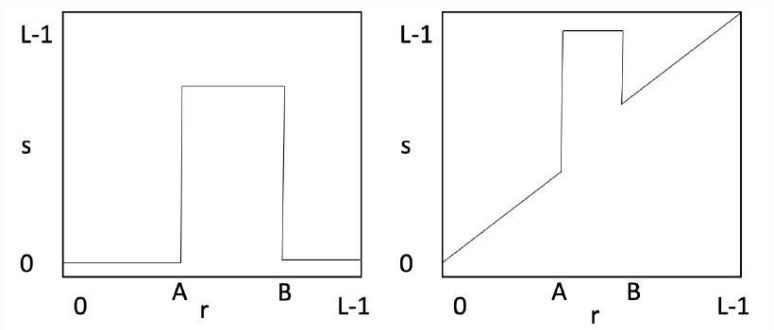


Figure 1.6.3: Example mapping functions for Intensity Level Slicing. The mapping on the left zeros out all pixels that have intensities below A ($r < A$) and above B ($r > B$). The mapping on the right brightens pixels with intensities between A and B ($A \leq r \leq B$).

Histogram Equalization

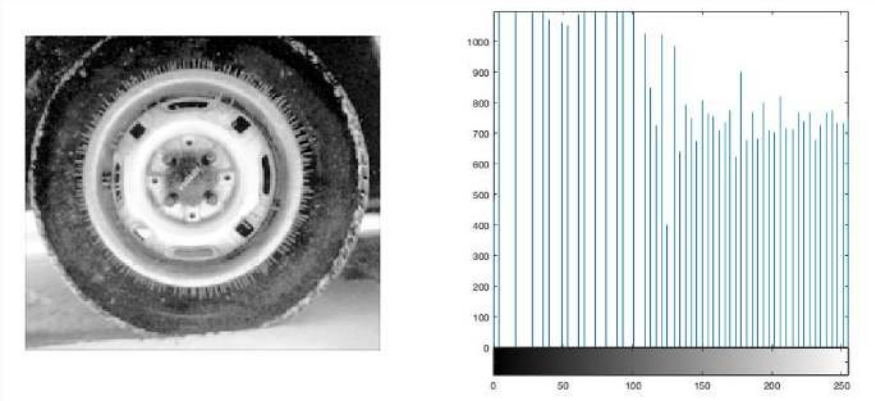


Figure 1.6.3: Tire image after histogram equalization with a histogram of its intensities.

The histogram equalization operation tries to change the pixel distribution so that it is as close as possible to a uniform distribution. This means that all gray levels from 0 to $2^B - 1$ (where B is the number of bits) are approximately equally likely. The effect is to widen the effective dynamic range of the image intensity. The results for this operation are shown in Figure 4 for the tire.tif image. Compare this to the original image in Figure 1. Note that the histogram extends across most gray values with approximately equal numbers of occurrences. MATLAB code for this operation is (`X` is the original image):

```
Xeq = histeq(X);
figure; subplot(121); imshow(Xeq)
subplot(122); imhist(Xeq)
```

Lab Exercises

1. Load the image “pout.tif” into MATLAB and compute its histogram. Include the histogram and image in your lab report.
    1. What is the range of the pixel intensity values in the original image?
    2. Give two methods for increasing the range of the image intensity from Section 4 above. Describe and implement the methods and show the results of the operations in your lab report. Compare the results in both cases, does one method work better than the other?
2. Load the built-in MATLAB image “coloredChips.png.” You will need to convert the image to grayscale. Use the methods described in Section 4 to isolate the round chips in the image by filtering the image intensity. The goal is to pre-process the image to remove the background so that a machine vision system can easily count the chips on the table. In your lab report, describe what steps you took to analyze the image and subtract the background. Include your MATLAB script and the stages of your processing pipeline. Display the image after each processing step.

Report Checklist

Be sure the following are included in your report.

1. Section 3.3: answer the three pre-lab questions
2. Section 5: answer the questions and complete the exercises in 1(a) and 1(b)
3. Section 5: complete and describe your approach to isolating the chips in 2

Reference

9 R.C. Gonzalez and R.E. Woods *Digital Image Processing*, Addison Wesley, 1993

"MATLAB Image Processing Toolbox Documentation",' *The Mathworks*, https://www.mathworks.com/help/images/index.html?s_tid=CRUX_lftnav

Jackson,Jeff Notes from ECE 482, University of Alabama,
jjackson.eng.ua.edu/courses/ece482/lectures/LECT01-2.pdf
jjackson.eng.ua.edu/courses/ece482/lectures/LECT05-2.pdf

This page titled 1.6: Introduction to Digital Images is shared under a CC BY-NC 4.0 license and was authored, remixed, and/or curated by Andrew K. Bolstad & Julie A. Dickerson (Iowa State University Digital Press) via source content that was edited to the style and standards of the LibreTexts platform.

1.7: Filtering Digital Images

Learning Objectives

By the end of this lab, students will be able to:

1. Describe spatial frequency in 2D digital images.
2. Design and implement linear, spatially invariant (LSI) filters using two-dimenional convolutional windows.
3. Use the properties of LSI filters to implement the Canny Edge Detection Protocol for images.

Overview

In this lab, we introduce linear sliding window filtering of images. The filters work in two dimensions to enhance the image in ways such as averaging out noise or emphasizing edges.

Pre-Lab Reading and Exercises

Image Filtering and Spatial Frequency

Image analysis deals with techniques for extracting information from images. The first step is generally to segment an image. Segmentation divides an image into its constituent parts or objects. For example, in a military air-to-ground target acquisition application one may want to identify tanks on a road. The first step is to segment the road from the rest of the image and then to segment objects on the road for possible identification. Segmentation algorithms are usually based on one of two properties of gray-level values: discontinuity and similarity. For discontinuity, the approach is to partition an image based on abrupt changes in gray level. Objects of interest are isolated points, lines, and edges.

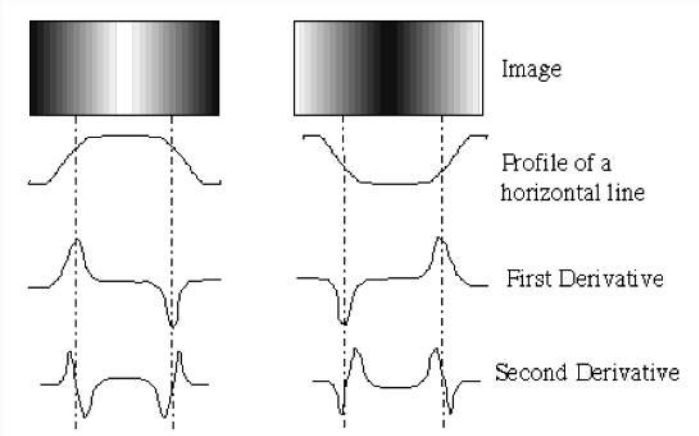


Figure 1.7.1: Synthetic images with horizontal pixel intensity, first derivative of intensity, and second derivative of intensity.

An edge is the boundary between two regions with relatively distinct gray-level properties. The idea underlying most edge-detection techniques is the computation of a relative derivative operator. Figure 1.7.1 illustrates this concept. The picture on the left shows an image of a light stripe on a dark background, the gray-level profile of a horizontal scan line, and the first and second derivatives of the profile. The first derivative is positive when the image changes from dark to light and zero where the image is constant. The second derivative is positive for the part of the transition associated with the dark side of the edge and negative for the transition associated with the light side of the edge. Thus, the magnitude of the first derivative can be used to detect the presence of an edge in the image and the sign of the second derivative can be used to detect whether a pixel lies on the light or dark side of the edge.

Sliding Window Filters

One way of filtering is to first filter the rows with a one-dimensional filter and then filter the columns with a one-dimensional filter. Alternatively, two dimensional filters can be applied to the neighborhood of each pixel to filter a signal. Filtering can help remove

noise from an image by averaging out the effects of random noise fluctuations. High pass filters help detect edges and sudden spatial changes in an image.

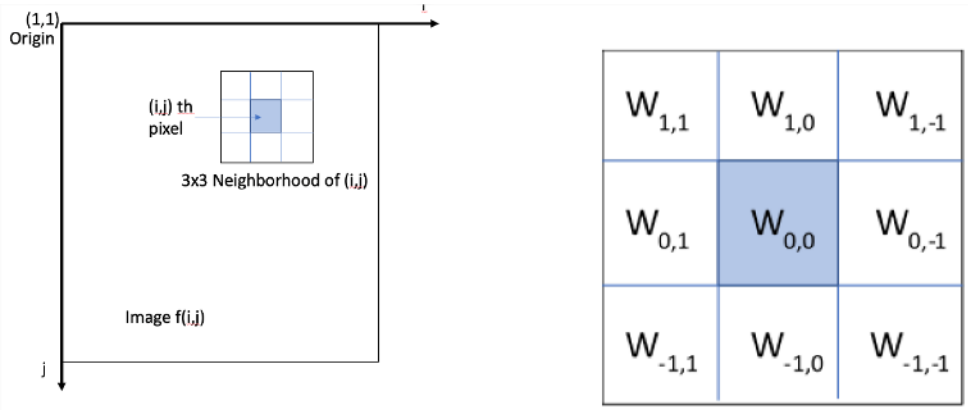


Figure 1.7.2: Spatial filtering with a sliding window; (left) three by three neighborhood of pixel (i,j); (right) three by three spatial mask.

The operations take place in the *spatial domain* and operate directly on pixel values. The general form for operations is: $y(i,j) = T[x(i,j)]$ where $x(i,j)$ is the input image, $y(i,j)$ is the filtered output image, T is an operator on x defined over a neighborhood of point (i,j). The operator, T, defines the size of the neighborhood around each pixel. The neighborhood is usually rectangular, centered on (i,j) and is much smaller than the image. This neighborhood is moved around to each pixel, hence the name sliding window or spatial mask. Use of spatial masks for filtering is called spatial filtering and the masks can be linear or nonlinear. As a note, the linear filter is equivalent to a two-dimensional convolution of the image with the filter.

A three by three linear filter can be written in matrix form as:

$$\begin{bmatrix} w_{1,1} & w_{1,0} & w_{1,-1} \\ w_{0,1} & w_{0,0} & w_{0,-1} \\ w_{-1,1} & w_{-1,0} & w_{-1,-1} \end{bmatrix}$$

The output of applying a three by three mask to image x is given by the two-dimensional convolution:

$$\begin{aligned} y(i,j) &= \sum_{k=-1}^{1}\sum_{l=-1}^{1} w_{k,l}x(i-k,j-l) \\ &= w_{1,1}x(i-1,j-1) + w_{1,0}x(i-1,j) + w_{1,-1}x(i-1,j+1) \\ &\quad + w_{0,1}x(i,j-1) + w_{0,0}x(i,j) + w_{0,-1}x(i,j+1) \\ &\quad + w_{-1,1}x(i+1,j-1) + w_{-1,0}x(i+1,j) + w_{-1,-1}x(i+1,j+1) \end{aligned}$$

This masking process continues until all pixels in the image are covered. The standard practice is to create a new image with the new values. For pixels near the boundary of the image, the output may be computed using partial neighborhoods or by padding the input appropriately by adding in repeated edge rows/columns or by a frame of black or white pixels.

These types of spatial filters are linear and spatially invariant (LSI). They can also be interpreted in the frequency domain. Lowpass filters attenuate (or eliminate) high frequency components characterized by edges and sharp details in an image with the net effect of blurring the image. Highpass filters attenuate (or eliminate) low frequency components such as slowly varying characteristics. The overall effect is a sharpening of edges. Bandpass filters attenuate (or eliminate) a given frequency range of frequencies. This is primarily used for image restoration.

Technical Notes for Implementing Filter Masks

In this lab, the MATLAB routine `imfilter` will be used: `Y = imfilter(X,h);` where X is the image to be filtered, h is the filter in matrix form, and Y is the filtered image. A simple example for a three by three averaging filter is given below:

```
>>X=imread('moon.tif');
>>h = ones(3)/9;% or [1,1,1;1 1 1;1 1 1]/9;
>>Y = imfilter(X,h);
```

There are two things to keep in mind with `imfilter`. First, it returns an image in the same format as the input. This means that if, for a uint8 image, the function gets a non-integer value, it rounds it to the closest integer. Second, it clips pixel values outside of the range [0, 255] to 0 or 255. These are non-linear operations which effect whether the LSI operations can be done in any order. For pixels near the boundary of the image, the output may be computed using partial neighborhoods or by padding the input appropriately by adding in repeated edge rows/columns or by a frame of black or white pixels. The default in `imfilter` is to assume a black border. Details can be found by reading about padding options on the help page.

Pre-Lab Questions

1. Which part of the image in Figure 1.7.3 has the lowest spatial frequency: specify top, bottom, left, right, middle, or a corner.

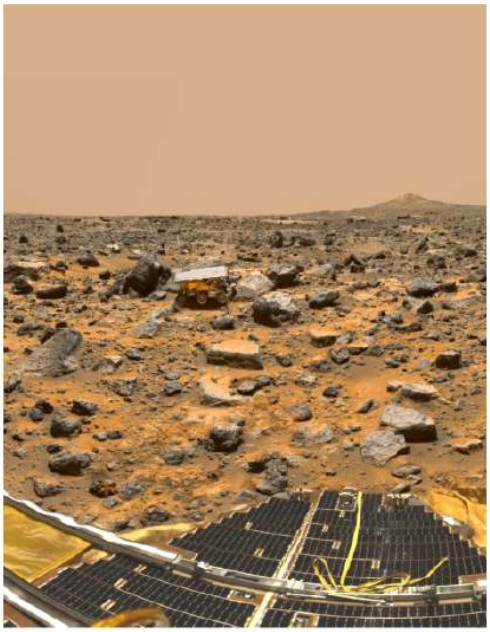


Figure 1.7.3: Image for question 1.

2. Compute the spatial period of the image below in terms of pixels/cycle. Give a numerical answer.

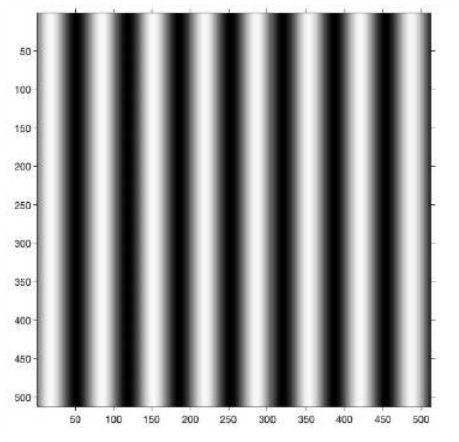


Figure 1.7.3: Image for question 2.

3. Given the 2D filter mask below and the intensity values from an image, compute the output for pixel at location $(3, 3)$ in the

image.

Lab Exercises

This part of the lab will be done using the three images that were provided with the lab. They are called I, I_g, and I_{sp}. For each piece of the lab, you will need to display the entire image before and after filtering. The command for this is:

```
>> imshowpair(I,filtI,`montage')
```

You will also need to display a close up of parts of the image that illustrate the filter effect well. To do this, it helps to display the image with the pixels numbered to help select the proper subimage. If the routine `imshow` does not do this automatically, then use the following code to display image I, for example:

```
>> RI = imref2d(size(I)); figure; imshow(I,RI);
```

Load the Matlab file *matlab.mat* with the images.

Low-Pass Spatial Filtering with a Moving Average Filter

A simple three by three or $N = 3$ averaging filter has the form:

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Write out the 2D filter for the $N = 5$ case. For averaging filters, the sum of the weights over the filter should be equal to 1. Use filters with dimensions $N = 5$, 9, and 13 to filter the image I. What is the effect of increasing the filter size in parts of the image with small details? Include your output images in your report.

High-Pass Spatial Filtering

The image filters shown below respond more strongly to lines of different directions in an image. The first mask responds very strongly to horizontal lines with a thickness of one pixel. The maximum response occurs when a line passed through the middle row of the mask. The direction of a mask can be established by noting that the preferred direction is weighted with a larger coefficient than the other possible directions.

(a)			(b)			(c)			(d)		
-1	-1	-1	-1	-1	2	-1	2	-1	2	-1	-1
2	2	2	-1	2	-1	-1	2	-1	-1	2	-1
-1	-1	-1	2	-1	-1	-1	2	-1	-1	-1	2

Use the image I. Filter the image using the four filters above. Describe your results for each case. Which edges were highlighted in each case? The MATLAB commands for the first case are given below.

```
>> wa=[-1,-1,-1;2,2,2;-1,-1,-1];
>> I_a=imfilter(I,wa);
```

Display your results by plotting the four output images together. Comment on your results and what edges were accented in each case.

Filtering Noise

Noise often has high spatial frequency so that it looks like salt and pepper in the picture. Most objects in the image have lower spatial frequencies so we can often improve the image by low pass filtering with an averaging filter. This technique works well if the noise does not vary that much from the image. Speckle or impulsive noise is very extreme noise that looks like white or black spots on an image. This section looks at how averaging filters affect the image.

1. [sec:nfilt1] Display the images I_g and I_{sp} in grayscale. These images have been corrupted by two different types of noise. You will use an averaging filter to remove the noise in this image. Filter each image with a 3x3 averaging filter and a 7x7 averaging filter. Display the results in image pairs. Discuss the results. Assess the trade-off between noise reduction and image resolution (how clear objects are in the image).

   Compare the results for filtering different types of noise. Does the filtering work better for one type of noise? If so, why?

2. [sec:nfilt2] [sec:imfilt] Using the images I_g and I_{sp}, filter each image with the high pass filters (a) and (c) from Section 4.2. Display the results in image pairs. Discuss the results focusing on how clear the edges in each direction appear.

   Compare the results for filtering different types of noise. Does the filtering work better for one type of noise? If so, why?

Filtering Edges in Noise

Edge detection is a key processing step for finding objects with computer vision systems. Unfortunately, noise can disrupt finding edges, so a commonly used strategy is to first blur the image to remove noise, then to use a high pass filter to find edges. Most practical edge detection operators use multi-stage algorithms to detect a wide range of edges in images. One example is Canny Edge Detection, developed by John F. Canny in 1986. The Canny Edge Detector first smooths the image to blur the noise to prevent false detections. Then, it uses a set of high pass filters to detect edges and estimate their direction. Further processing filters and thins the edges. In this lab, we implement the first two steps.

1. [sec:nefilt1] **Smoothing:** Apply a low pass filter to smooth the image to remove noise to prevent false detections caused by noise. The typical filter used by this function is a Gaussian filter kernel which smooths the edges of the filter. Use the filter parameters below to filter image I_g:

   ```
   >> hgauss = [2 4 5 4 2; 4 9 12 9 4; 5 12 15 12 5; ...
            4 9 12 9 4; 2 4 5 4 2]/159;
   ```

   Display the result and comment on what you see in the image.

2. [sec:nefilt2] **Edge Detection and Intensity Gradient:** An edge in an image may point in a variety of directions, so the Canny algorithm uses filters to detect horizontal, vertical, and diagonal edges in the smoothed image. Use the Sobel Edge Detector filters given below on the result from the previous step with each filter separately to get images filtered in the x and y directions, G_x and G_y.

(a)			(b)		
-1	-2	-1	-1	-	1
0	0	0	-2	0	2
1	2	1	-1	0	1

Finally, combine the two filtered images to compute the intensity gradient and local direction at each pixel in the image:

$$G = \sqrt{G_x^2 + G_y^2}$$
$$\theta = \text{atan2}(G_y, G_x)$$

Display the intensity gradient image, G, using the command `imshow(G,[0,255])` and comment on your results from Section 4.3 Part 1 above. Are the edges clearer in one case or the other?

Thought Question

What will happen if the order of processing for an LSI low pass filter followed by an LSI high-pass filter is reversed for edge detection? Explain if order matters or not.

Report Checklist

Be sure the following are included in your report.

1. Section 3: answer the three pre-lab questions
2. Section 4.1: write out 5x5 filter
3. Section 4.1: report on filter effects for $N = 5$, 9, and 13; include images
4. Section 4.2: describe results of applying the four filters, answer related questions; include images
5. Section 4.3, Part 1: answer questions about noise reduction vs. resolution; include images
6. Section 4.3, Part 2: answer questions about edge enhancement; include images
7. Section 4.4, Part 1: display filtered image and comment
8. Section 4.4, Part 2: display gradient and direction images; compare to Section 4.3, Part 2
9. Section 4.5: answer the thought question

References

R.C. Gonzalez and R.E. Woods, *Digital Image Processing*, Addison Wesley, 1993

"MATLAB Image Processing Toolbox Documentation", *The Mathworks*, https://www.mathworks.com/help/images/index.html?s_tid=CRUX_lftnav

"Canny Edge Etector", *Wikipedia*, https://en.Wikipedia.org/wiki/Canny_edge_detector

J. Canny, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679-698, Nov. 1986

Bill Green, "Canny Edge Detection Tutorial", https://web.archive.org/web/20160324173252/http:/dasl.mem.drexel.edu/alumni/bGreen/www.pages.drexel.edu/_weg22/can_tut.html

Prem K. Kalra, "Canny Edge Detection", https://web.archive.org/web/20150421090938/http:/www.cse.iitd.ernet.in/~pkalra/csl783/canny.pdf

NASA Image and Video Library https://images.nasa.gov/

This page titled 1.7: Filtering Digital Images is shared under a CC BY-NC 4.0 license and was authored, remixed, and/or curated by Andrew K. Bolstad & Julie A. Dickerson (Iowa State University Digital Press) via source content that was edited to the style and standards of the LibreTexts platform.

Detailed Licensing

Overview

Title: Open Signals and Systems Laboratory Exercises (Bolstad and Dickerson)

Webpages: 21

Applicable Restrictions: Noncommercial

All licenses found:

- CC BY-NC 4.0: 76.2% (16 pages)
- Undeclared: 23.8% (5 pages)

By Page

- Open Signals and Systems Laboratory Exercises (Bolstad and Dickerson) - *CC BY-NC 4.0*
  - Front Matter - *CC BY-NC 4.0*
    - TitlePage - *CC BY-NC 4.0*
    - InfoPage - *CC BY-NC 4.0*
    - Table of Contents - *Undeclared*
    - Licensing - *CC BY-NC 4.0*
    - Acknowledgments - *Undeclared*
    - Introduction - *Undeclared*
  - 1: Labs - *Undeclared*
    - 1.1: MATLAB Basics - *CC BY-NC 4.0*
    - 1.2: Tuning Forks - *CC BY-NC 4.0*
    - 1.3: Impulse Response and Auralization - *CC BY-NC 4.0*
    - 1.4: Frequency Response - Notch and Bandpass Filters - *CC BY-NC 4.0*
    - 1.5: Fourier Series - *CC BY-NC 4.0*
    - 1.6: Introduction to Digital Images - *CC BY-NC 4.0*
    - 1.7: Filtering Digital Images - *CC BY-NC 4.0*
  - Back Matter - *CC BY-NC 4.0*
    - Index - *CC BY-NC 4.0*
    - Glossary - *CC BY-NC 4.0*
    - Detailed Licensing - *CC BY-NC 4.0*
    - Detailed Licensing - *Undeclared*

Detailed Licensing

Overview

Title: Open Signals and Systems Laboratory Exercises (Bolstad and Dickerson)

Webpages: 21

Applicable Restrictions: Noncommercial

All licenses found:

- CC BY-NC 4.0: 76.2% (16 pages)
- Undeclared: 23.8% (5 pages)

By Page

- Open Signals and Systems Laboratory Exercises (Bolstad and Dickerson) - *CC BY-NC 4.0*
  - Front Matter - *CC BY-NC 4.0*
    - TitlePage - *CC BY-NC 4.0*
    - InfoPage - *CC BY-NC 4.0*
    - Table of Contents - *Undeclared*
    - Licensing - *CC BY-NC 4.0*
    - Acknowledgments - *Undeclared*
    - Introduction - *Undeclared*
  - 1: Labs - *Undeclared*
    - 1.1: MATLAB Basics - *CC BY-NC 4.0*
    - 1.2: Tuning Forks - *CC BY-NC 4.0*
    - 1.3: Impulse Response and Auralization - *CC BY-NC 4.0*
    - 1.4: Frequency Response - Notch and Bandpass Filters - *CC BY-NC 4.0*
    - 1.5: Fourier Series - *CC BY-NC 4.0*
    - 1.6: Introduction to Digital Images - *CC BY-NC 4.0*
    - 1.7: Filtering Digital Images - *CC BY-NC 4.0*
  - Back Matter - *CC BY-NC 4.0*
    - Index - *CC BY-NC 4.0*
    - Glossary - *CC BY-NC 4.0*
    - Detailed Licensing - *CC BY-NC 4.0*
    - Detailed Licensing - *Undeclared*