Open Signals and Systems Laboratory Exercises

Second Edition

Aaron J. Fonseca
Julie A. Dickerson

Iowa State University
Digital Press

Open Signals and Systems Lab Exercises

SECOND EDITION

Aaron Fonseca and Julie Dickerson

IOWA STATE UNIVERSITY DIGITAL PRESS

Ames

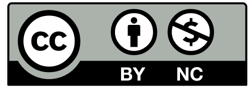
May 21, 2024

This work is published by:

Iowa State University Digital Press

701 Morrill Rd, Ames, Iowa 50011, United States

https://www.iastatedigitalpress.com/



© 2024 Aaron Fonseca and Julie Dickerson. This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License, except where otherwise noted.

The publisher is not responsible for the content of any third-party websites. URL links were active and accurate at time of publication.

DOI: https://doi.org/10.31274/isudp.2024.155

Iowa State University is located on the ancestral lands and territory of the Baxoje (bah-kho-dzhe), or Ioway Nation. The United States obtained the land from the Meskwaki and Sauk nations in the Treaty of 1842. We wish to recognize our obligations to this land and to the people who took care of it, as well as to the 17,000 Native people who live in Iowa today.

Contents

Acknowledgments **xi**

Preface **xiii**

1 MATLAB Basics **1**

1.1 Introduction 1
1.2 The MATLAB Window 2
1.3 Variables, Results, and Outputs 3
1.4 Mathematical Operations in MATLAB 5
1.4.1 Built-in Functions in MATLAB 6
1.4.2 Getting Help for MATLAB Functions 7
1.4.3 Functions with Multiple Parameters 7
1.4.4 Complex Numbers in MATLAB 8
1.5 Basics of Vectors and Matrices 9
1.5.1 Review of Vectors 9
1.5.2 Review of Matrices 10
1.5.3 Vectors and Matrices in MATLAB 10
1.5.4 Generating Vectors and Matrices 13
1.5.5 Operations on Vectors and Matrices 17
1.5.6 Operations Between Vectors/Matrices 19
1.5.7 Functions applied to Vectors and Matrices 20
1.6 Basic Plotting 22
1.6.1 Changing the Viewing Region 25
1.6.2 Plot-like Functions 26
1.7 Utilizing Scripts 27
1.8 Advanced Plotting 30
1.8.1 Multiple Figures 30
1.8.2 Subplots 32

1.8.3 Multiple Plots on the Same Axes . . . 34
1.8.4 Plotting Complex-valued Functions . . . 36
1.9 Post-Lab Questions . . . 37
1.10 Report Checklist . . . 40
1.R References . . . 42

2 Sampled Sounds, Echo and Reverberation **45**
2.1 Introduction . . . 46
2.2 MATLAB Background . . . 46
2.2.1 Indexing and Slicing Vectors/Matrices . . . 46
2.2.2 Slicing Vectors . . . 48
2.3 Processing Digital Audio Files . . . 53
2.3.1 Working with Audio in MATLAB . . . 54
2.4 Modeling Echo and Reverberation . . . 58
2.4.1 Echo . . . 58
2.4.2 Reverberation . . . 60
2.4.3 Simulink . . . 61
2.4.4 The Echo System . . . 61
2.4.5 The Reverb System . . . 63
2.4.6 Singing Rounds . . . 65
2.4.7 Creating a Realistic Echo Subsystem . . . 65
2.5 Post-Lab Questions . . . 66
2.6 Report Checklist . . . 67
2.7 Acknowledgments . . . 68
2.R References . . . 69

3 Introduction to CyDAQ and DAD **71**
3.1 Introduction . . . 72
3.2 Connections and Setup . . . 74
3.3 CyDAQ Software . . . 75

3.4 WaveForms Software . . . 76
3.4.1 The FFT View . . . 81
3.4.2 Measuring Tools . . . 83
3.5 Capturing Data with the CyDAQ . . . 84
3.6 Practical Limitations of the CyDAQ . . . 89
3.7 Post-Lab Questions . . . 90
3.8 Report Checklist . . . 92
3.9 Acknowledgments . . . 92
3.R References . . . 93

4 CyDAQ Impulse Responses **95**
4.1 Introduction . . . 96
4.2 Characterizing Systems with Impulse Response . . . 96
4.2.1 Playing Audio with the CyDAQ . . . 98
4.2.2 Filtering with the CyDAQ . . . 99
4.2.3 Measuring the CyDAQ's Impulse Response . . . 99
4.2.4 Using the Impulse Response as a Filter . . . 102
4.3 Auralization . . . 102
4.4 Report Checklist . . . 104
4.5 Acknowledgments . . . 105
4.A Hardware Ports . . . 106
4.B Convolve Audio MATLAB Script . . . 107
4.R References . . . 109

5 Frequency Response and System Identification **111**
5.1 Introduction . . . 112
5.2 Background on Frequency Response . . . 112
5.3 System Identification using Frequency Response . . . 113
5.3.1 Manual Measurements in WaveForms . . . 114
5.3.2 Tabulate Your Results . . . 116

5.3.3 Automated Measurements . . . 117
5.4 Comparing Results in MATLAB . . . 118
5.5 Report Checklist . . . 119
5.6 Acknowledgments . . . 120
5.A Hardware Ports . . . 120
5.R References . . . 121

6 Fourier Series Part I **123**
6.1 Introduction . . . 123
6.2 MATLAB Background . . . 124
6.2.1 Matrix Concatenation . . . 124
6.2.2 Combining Concatenation with Array Slicing . . . 125
6.2.3 Successive Concatenation . . . 125
6.3 Fourier Series and Musical Instruments . . . 126
6.3.1 Fourier Series Definition . . . 126
6.3.2 Fourier Series of a Triangle Wave . . . 126
6.3.3 Relationship to Musical Instruments . . . 128
6.3.4 Riemann Approximation of Fourier Series Coefficients . . . 129
6.4 Computing Fourier Series Coefficients using MATLAB . . . 130
6.4.1 Approximating the Fourier Series of a Triangle Wave . . . 131
6.4.2 Re-synthesizing a Square Wave . . . 135
6.4.3 Re-synthesizing a Piano . . . 135
6.5 Report Checklist . . . 138
6.6 Acknowledgements . . . 140
6.A Generate Triangle Wave Function . . . 140
6.B Generate Square Wave Function . . . 141
6.C Compute Fourier Series Coefficients Function . . . 141
6.D Compute Fourier Series Coefficient Function . . . 142
6.E Synthesize Fourier Series Function . . . 143
6.R References . . . 144

7 Fourier Series Part II **147**

7.1 Introduction 147

7.2 MATLAB Background 148

7.3 Background 149

7.4 Re-synthesizing Instruments 150

7.4.1 Isolating Slices from Recordings 150

7.4.2 Re-synthesizing the Instrument 156

7.4.3 Re-synthesize two more Instruments 157

7.4.4 Synthesize a Chromatic Scale 158

7.5 Report Checklist 158

7.6 Acknowledgements 159

7.A Plot Pitch Function 160

7.B Cut Sample Function 161

7.C Compute Fourier Series Coefficients Function 161

7.D Compute Fourier Series Coefficient Function 162

7.E Synthesize Fourier Series Function 163

7.R References 164

8 Introduction to Digital Images **167**

8.1 Introduction 167

8.2 Images in MATLAB 168

8.2.1 Monochromatic Images 168

8.2.2 Displaying and Exporting Images 169

8.2.3 An Image of the Moon 170

8.3 Image Manipulation in MATLAB 171

8.3.1 Image Histograms 171

8.3.2 Contrast Stretching and Intensity Level Slicing 172

8.3.3 Histogram Equalization 173

8.4 Isolating Areas of Interest 174

8.5 Report Checklist 175

8.A Level Slice Function 176
8.B Contrast Stretch Function 177
8.R References 179

9 Digital Image Processing **181**
9.1 Introduction 181
9.2 Background 181
9.2.1 Spatial Frequency 181
9.2.2 Regions of Spatial Frequency 182
9.2.3 Computing Spatial Period 183
9.2.4 Sliding Window Filters 183
9.2.5 Manually Computing the Value of a Filtered Pixel 185
9.2.6 MATLAB Background 185
9.3 Filtering Images 186
9.3.1 Low-Pass Spatial Filtering Using a Moving Average Filter 187
9.3.2 High-pass Spatial Filtering 187
9.3.3 Filtering Noise 188
9.3.4 Isolating Edges from Noisy Images 189
9.4 Report Checklist 190
9.R References 192

10 Pulse-width Modulation and Filter Design **193**
10.1 Introduction 194
10.2 Pulse-width Modulation 194
10.3 Decoding Pulse-width Modulation 196
10.4 Encoding Pulse-width Modulation 198
10.4.1 MATLAB Simulation 199
10.4.2 PWM Circuit 201
10.5 Report Checklist 204
10.6 Acknowledgments 205

10.A Hardware Ports 205
10.B PWM Simulation MATLAB Script 206
10.R References 209

11 Sampling and Aliasing **211**
11.1 Introduction 211
11.2 Background 212
11.2.1 Sampling in the Frequency Domain 212
11.2.2 The Discrete Fourier Transform 213
11.3 Effects of Sampling 213
11.3.1 Sampling Near Nyquist 214
11.3.2 Upsampling for Reconstruction 216
11.3.3 Aliasing 217
11.4 Report Checklist 218
11.5 Acknowledgments 219
11.A Plot Spectrum MATLAB Code 219

12 Demodulation with a Software Defined Radio **221**
12.1 Introduction 221
12.2 The SdrSharp Software 222
12.2.1 Using the Software 222
12.3 Demodulating with SdrSharp 226
12.3.1 FM Demodulation 226
12.3.2 AM Demodulation 228
12.4 AM Demodulation with MATLAB 229
12.4.1 Demodulating using Envelope Detection 229
12.4.2 Demodulating using Coherent Detection 230
12.5 Report Checklist 230
12.6 Acknowledgements 231
12.A AM Coherent Demodulation Standalone Script 231

12.B AM Envelope Demodulation Standalone Script 233
12.C AM Coherent Detection Function 234
12.D AM Envelope Detection Function 235
12.E SDR Read Function 236
12.R References 237

Acknowledgments

The authors are deeply indebted to the following individuals, without whom this Second Edition would not be possible:

Andrew Bolstad—An original author of the first edition of the lab exercises. After the publication of the first edition, he collaborated with students and the Electronics Technology Group to conceive and draft additional exercises that incorporated new pieces of lab equipment, such as the Cyclone Data Acquisition board.

Taylor Burton—An EE224 Spring 2020 student who conceived the idea of Fourier Series-based lab exercises that involved synthesizing instruments.

Mathew Post—An engineer with Iowa State University's Engineering Technology Group. He is the inventor and project manager of the *Cyclone Data Acquisition* (CyDAQ) hardware platform. He provided hardware troubleshooting and offered insight into the capabilities of the CyDAQ throughout the development of these lab exercises.

Isaac Rex—A student who helped engineer the *Cyclone Data Acquisition* (CyDAQ) board for his senior design project in Fall 2020. He conceived and drafted all exercises involving the CyDAQ (specifically the Introduction to CyDAQ, Impulse Response, Frequency Response, Filtering, and Sampling exercises).

Connor Ryan—A student involved with the Electronic Technology Group. He helped draft the initial version of the Software Defined Radio/Communications exercise.

Preface

The lab exercises for Iowa State University's EE224 *Signals and Systems I* course are an evolving work. In 2021, Andrew Bolstad and Julie Dickerson published the first edition of these exercises with support from a 2019 Miller Open Education Mini-grant. While this first edition provided a basic introduction to programming in MATLAB, it presumed existing familiarity with MATLAB and required substantial supplementary MATLAB recitations to enable student success. There was a desire to update the lab manuals such that they could "stand on their own" and cover the full breadth of MATLAB background necessary to succeed.

The labs conducted in the semesters following the first edition's publication also saw the integration of new lab equipment into the exercises, including the *Cyclone Data Acquisition Board* (CyDAQ), the *Digilent Analog Discovery 2* (DAD), and the *Realtek RTL2832U Software Defined Radio* (RTL-SDR). Of these, the most substantially integrated component was the CyDAQ, a hardware exploration platform and configurable filter developed in-house at Iowa State University. Lab manuals for the CyDAQ integrated lab exercises were drafted by Andrew Bolstad and Isaac Rex but were distinct in style and did not always align with the first edition's lab exercises.

Beginning in 2023, Aaron Fonseca and Julie Dickerson set out to standardize and improve the EE224 lab exercises in use. The goals of these updates were:

(a) To provide a thorough coverage of the MATLAB concepts needed to complete each exercise.

(b) To incorporate hardware descriptions and explanations into the exercises that had integrated new lab equipment.

In 2024 these updates were completed and the resulting collection of lab exercises was released as an Open Educational Resource (OER) through the Iowa State University Digital Press as the Second Edition of the *Open Signals and Systems Lab Exercises*.

MATLAB Basics

Introduction to MATLAB

Overview

In this lab exercise, students will be introduced to the MATLAB software. Students will input commands into MATLAB and record the resulting textual and graphical outputs.

Learning Objectives

By the end of this lab exercise, students will be able to do the following in MATLAB:

1. Evaluate mathematical expressions.
2. Manually construct variables containing vectors and matrices.
3. Generate vectors and matrices consisting of a single repeated value.
4. Generate vectors of sequential values.
5. Retrieve the size or dimensionality of matrices and vectors.
6. Apply functions as a map to each element of a vector or matrix.
7. Construct mathematical expressions containing scalars, vectors, and matrices.
8. Differentiate between element-wise and matrix operations.
9. Contruct plots of functions.
10. Construct figures with multiple constituent subplots.
11. Construct plots depicting multiple functions on the same axes.
12. Indentify errors caused by unintended retention of MATLAB program state.

1.1 Introduction

This lab exercise introduces the MATLAB computing environment. MATLAB is a handy tool for signals, systems, and many other computing tasks. This exercise briefly introduces fundamental concepts of MATLAB, such as vector/matrix initialization and generation and plotting signals. Future lab exercises will introduce new MATLAB content as necessary. Navigate to

your start menu and launch MATLAB. After loading the program, you should be greeted with the MATLAB main window.

1.2 The MATLAB Window

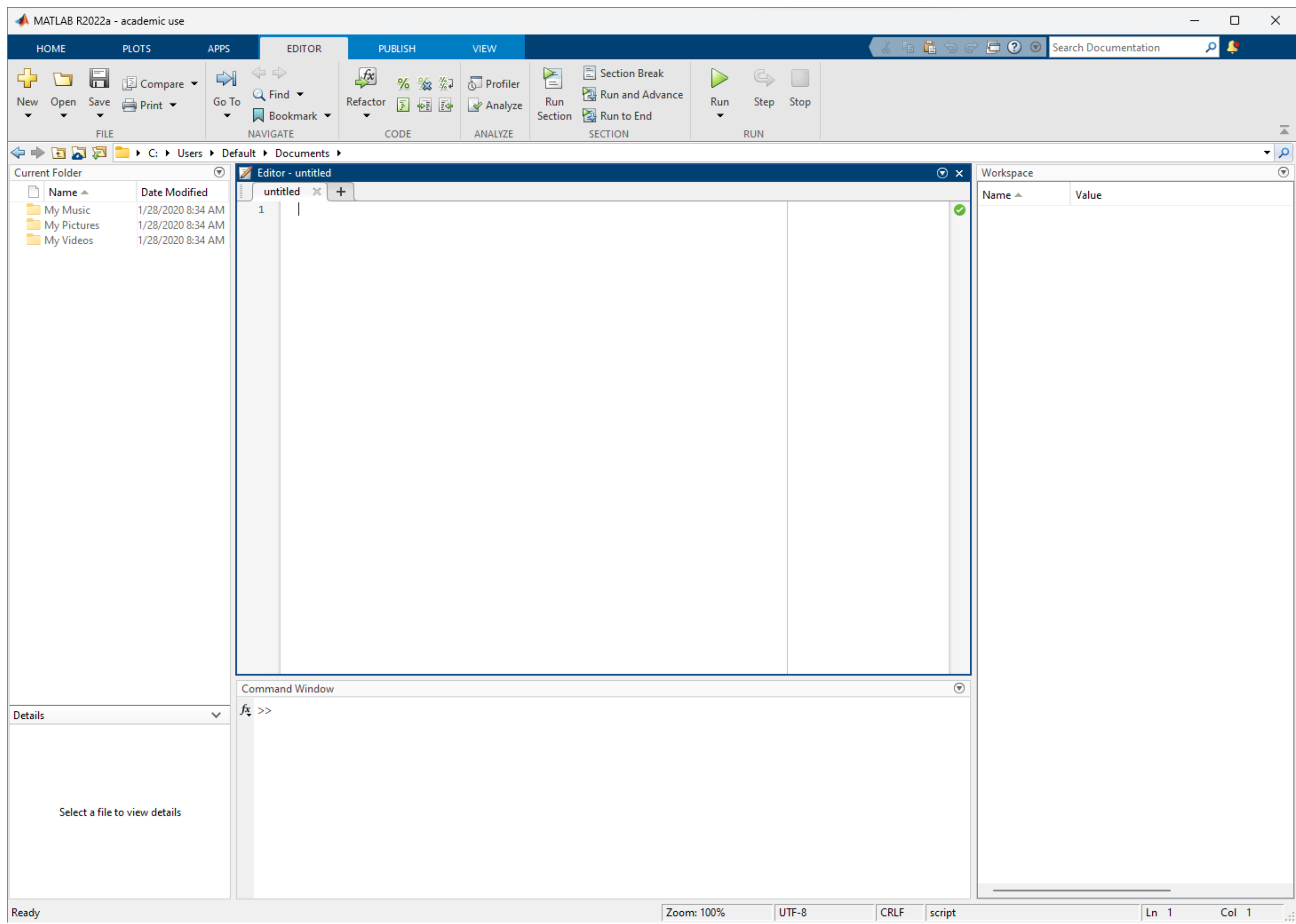


Figure 1.1: The MATLAB Default Screen

Upon launching MATLAB, you will be greeted with the *default screen* (depicted in Fig. 1.1). At the top-most portion of the *default screen*, you will see the **Upper Menu**. This is a notebook-style menu with the **Editor** tab selected. Just below the **Upper Menu** is the **Address Bar**. This gives the path of MATLAB's *current working directory*. The left-most portion of the *default screen* gives the **Current Folder View**. This displays the contents of MATLAB's *current working directory*. The center of the *default screen* gives the **Script Editor Window**. If you do not see the **Script Editor Window**, the keyboard shortcut `Ctrl+N` should open a new

Script Editor Window. Below the **Script Editor Window** is the **Command Window**. Finally, the right-most portion of the *default screen* gives the **Variable Workspace**.

Unlike C, C++ and Java—MATLAB is an interpreted language. This means that MATLAB code does not need to be compiled before it can be run. Instead, MATLAB maintains a *program state*, which can be manipulated via *commands*. These *commands* are interpreted after each new line in what is termed a read–eval–print loop (REPL).[1] But this is all still very abstract—let's dive right in with some hands-on examples.

1.3 Variables, Results, and Outputs

Navigate to the **Command Window** and select the input prompt (immediately following '>>') so that you can begin entering *commands*. Enter the command:

```
x = 2
```

MATLAB will return:

```
x =

    2
```

This *command* has just *assigned* the variable `x` to `2` and MATLAB printed output informing you it did so. Enter the command:

```
y = x + 2
```

MATLAB will return:

```
y =

    4
```

This command *evaluated* the mathematical expression `x + 2` and *assigned* the result to the variable `y`. It then printed output informing you it did so.

Now, let's do something slightly different. Enter the command:

```
x + 5
```

MATLAB will return:

```
ans =

    7
```

This command *evaluated* the mathematical expression `x + 5` and printed the result. Actually, MATLAB did more than print the result. It also assigned the result to a special variable called `ans`. Whenever you tell MATLAB to *evaluate* an expression and you do not explicitly assign a variable to the result of that expression, MATLAB will assign the result to the variable `ans`. While it is possible to manually assign the value of `ans` (e.g., `ans = 3`), it is highly discouraged.[2]

Now enter the following command (*note the semicolon at the end of the line*):

```
z = y + 3;
```

MATLAB will return... *nothing*. Does this mean that the *assignment* of `z` to `y + 3` didn't execute? No! When MATLAB commands are terminated by a semicolon (`;`) the *output* MATLAB produces is suppressed, but the command still executes. We can confirm that the command was successfully executed by entering:

```
z
```

Which produces the output:

```
z =

    7
```

Suppressing output with the semicolon (`;`) terminator is extremely common as the output of many MATLAB commands can be quite verbose. The other common use of the semicolon is to enable multiple commands to be executed on a single line. For example:

```
a = 1; b = 2;
```

will execute both assignment commands.

Now would be an excellent time to turn our attention to the **Variable Workspace**. If you've been following along closely, you should see results similar to those depicted in Fig. 1.2. The

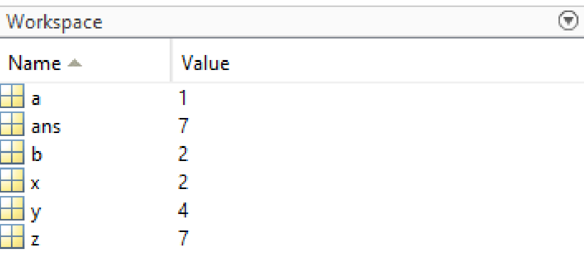


Figure 1.2: The Variable Workspace

Variable Workspace gives a listing of the *names* and *values* of the variables currently defined within MATLAB's *program state*. We can see all the variables we've defined as a consequence of the commands you've entered.

Now enter the command:

```
clearvars;
```

The **Variable Workspace** should now be empty. This command, `clearvars`, clears all the variables in MATLAB's *program state*.

Enter the command:

```
z = y + 3;
```

Comment on the result.

Rather than clearing the entire variable workspace, individual variables can be cleared using the `clear` command. For instance, to clear a variable, `y`, one would enter:

```
clear y;
```

To clear more than one variable—for instance, variables `y` and `z`—one could type:

```
clear y z;
```

1.4 Mathematical Operations in MATLAB

In addition to simple variable assignments, MATLAB can also *evaluate* mathematical expressions.[3] This includes the standard arithmetic operations: addition (`+`), subtraction (`-`), multipli-

cation (`*`) division (`/`), and exponentiation (`^`). For example, one could evaluate the expression:

$$x = \frac{(2 \cdot 3)^2 - 4}{2},$$

in MATLAB by entering the command:

```
x = ((2*3)^2 - 4)/2
```

Note that parentheses are used to indicate precedence for evaluation.

1.4.1 Built-in Functions in MATLAB

MATLAB provides a set of built-in mathematical *constants*[4] and *functions*.[5] These include:

- the constant `pi`,
- the trigonometric functions `sin`, `cos`, and `tan`,
- the inverse trigonometric functions `asin`, `acos`, and `atan`,
- the exponential function `exp`,
- the natural logarithm function `log`,
- the base-10 logarithm function `log10`,
- and many others.

For example, one could evaluate the expression:

$$y = e^{-2} \cos(3 \cdot 2\pi),$$

in MATLAB by entering

```
y = exp(-2) * cos(3*2*pi)
```

As an exercise to the reader, use MATLAB to evaluate the following expression:

$$z = \ln(2) \cdot \arctan(\pi/3).$$

Provide the MATLAB command and comment on the result in your lab report. *Hint*: The natural logarithm function is *not* called '`ln`' in MATLAB.

1.4.2 Getting Help for MATLAB Functions

MATLAB also provides many other useful built-in functions that you may encounter when browsing examples/code online. Should you ever come across a function you do not understand, use the `help` command to retrieve documentation for that function.

For instance, say you come across the function `gamma` and would like to learn what it does. Entering the command:

```
help gamma
```

retrieves the documentation for the function.

Using the `help` command is often more useful than googling a particular MATLAB function. This is because some functions differ in syntax between different versions of MATLAB and `help` will always return the documentation relevant to the version of MATLAB you're using. This is not meant to discourage you from googling MATLAB functions in search of information—just keep in mind that the information you locate online (even through the MathWorks website)[6] may not apply to your version of MATLAB.

1.4.3 Functions with Multiple Parameters

So far, all the MATLAB functions presented contained only one *parameter* (input). MATLAB also supports functions with multiple parameters. Calling functions with multiple parameters is best explained by example.

Consider the expression:

$$q = \binom{6}{4}, \tag{1.1}$$

where $\binom{6}{4}$ is the $n = 6$, $k = 4$ binomial coefficient (also called n-choose-k).[7] Clearly, the *binomial coefficient selection function* is a function of *two* parameters. In MATLAB, the binomial coefficient selection function is called `nchoosek`. We can evaluate Eq.(1.1) by entering the MATLAB command:

```
q = nchoosek(6, 4);
```

In general, multi-parameter functions are called by delimiting each parameter using commas (`,`).

1.4.4 Complex Numbers in MATLAB

In addition to *real*-valued numbers we've been working with so far, MATLAB is also capable of working with *imaginary*- and *complex*-valued numbers. MATLAB has the built-in constants: `i` and `j` which are both define to be the $\sqrt{-1}$. For example, the complex number:

$$a = 3 + 4\text{j},$$

can be assigned in MATLAB using the command:

```
a = 3 + 4*j
```

Newer versions of MATLAB also permit the multiplication sign (`*`) to be excluded:

```
a = 3 + 4j
```

Note that if your version of MATLAB supports this syntax, it only applies to coefficients of `i` and `j`. Outside these multiples, MATLAB does not permit any other instance of *implied multiplication*. In other words, do not omit `'*'` anywhere besides in the definition of complex numbers.

Often, built-in functions that take *real*-valued parameters can also take *complex*-valued parameters as well. We can leverage this fact to enable us to input complex numbers in polar form. For example, the polar-form complex number:

$$b = 2\ e^{1.4\text{j}}$$

can be defined in MATLAB by entering the command:

```
b = 2 * exp(1.4*j)
```

Comment on the output of the above command. Does MATLAB print the result in polar or rectangular form?

1.5 Basics of Vectors and Matrices

So far, we've been using MATLAB as a glorified calculator. Now, we will introduce *vectors* and *matrices* and begin leveraging MATLAB's full potential. The primary datatype in MATLAB are matrices and MATLAB was built and optimized to work with matrices. MATLAB contains a plethora of built-in routines precisely to manipulate matrices. This lab assumes you are familiar with *vectors* and *matrices*. Still, we will briefly review aspects of them here to establish a shared understanding and vocabulary.

1.5.1 Review of Vectors

In essence, a *vector* is an ordered sequence of numbers (real or complex). For instance, we could define the vector:

$$\mathbf{x} = \begin{bmatrix} 3, & 5, & 9 \end{bmatrix}. \tag{1.2}$$

In contrast to *vectors*, single instances of numbers (real or complex) are called *scalars*.

Vectors are said to have a *length* that corresponds to the *number of elements* in the vector. In the case of $\mathbf{x}$, the *length* is 3.

In MATLAB, vectors are also said to have an *orientation*. A vector can be a *row vector* or a *column vector*. In the case of $\mathbf{x}$: $\mathbf{x}$ is a *row vector* because it is illustrated horizontally. Column vectors, on the other hand, are illustrated vertically. For example:

$$\mathbf{y} = \begin{bmatrix} 3 \\ 5 \\ 9 \end{bmatrix} \tag{1.3}$$

is a *column vector*. The orientation of vectors has implications for algebraic operations between vectors—but we will not concern ourselves with that now. In the meantime, just know that the difference in orientation makes two different vectors *distinct* in MATLAB. In other words, $\mathbf{x} \neq \mathbf{y}$ despite both containing the same ordered sequence of numbers.

1.5.2 Review of Matrices

A *matrix* is a two-dimensional, ordered collection of numbers (real or complex). For example, we could define a matrix:

$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}. \tag{1.4}$$

Matrices are said to have a *size* (or *dimensionality*). In the case of $\mathbf{X}$ in Eq. (1.4), $\mathbf{X}$ is said to have a *size* of 3×3, where the first number is the *number of rows* the matrix has and the second number is the *number of columns*. For $\mathbf{X}$, the number of rows and columns are equal; in such a case, the matrix $\mathbf{X}$ is said to be *square*.

Matrices need not be square. Consider the matrix:

$$\mathbf{Y} = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}. \tag{1.5}$$

Clearly $\mathbf{Y}$ from Eq. (1.5) is *not* square. Comment on the *size* of $\mathbf{Y}$ in your lab report.

Unlike vectors, matrices do not have a concept of *orientation*. This concept is already encoded by the matrix's *size*. In fact, it is possible to conceive of an m-length *row vector* as a matrix whose size is $1 \times m$, and an n-length column vector as a matrix whose size is $n \times 1$. It is often beneficial to conceptualize all vectors as matrices; other times, it is benficial to maintain a mental distinction between the two.

1.5.3 Vectors and Matrices in MATLAB

The matrix $\mathbf{X}$ from Eq. (1.4) can be defined in MATLAB with the command:

```
X = [1, 2, 3; 4, 5, 6; 7, 8, 9]
```

Enter this command. MATLAB will print the output

```
X =

    1    2    3
```

```
    4     5     6
    7     8     9
```

As can be seen in the preceding example:

- the contents of a matrix are defined within square-brackets `[ ... ]`,
- the columns of a matrix are delimited by commas (`,`) and
- the rows are delimited by semi-colons (`;`).

Now enter the MATLAB commands to define/initialize:

- the matrix $\mathbf{Y}$ from Eq. (1.5),
- the row vector $\mathbf{x}$ from Eq. (1.2), and
- the column vector $\mathbf{y}$ from Eq. (1.3).

Provide each of these commands in your lab report. *Hint*: Remember that vectors can be thought of as one-dimensional matrices.

Matrix definitions in MATLAB are not constricted to only numbers—it is also possible to evaluate mathematical expressions within matrix definitions. For example, enter the command:

```
X = [2 + exp(0.1), 35; -70, -1 + tan(1.27)]
```

MATLAB produces the output

```
X =


    3.1052   35.0000
  -70.0000    2.2236
```

In addition to the matrix initialization syntax employed thus far, an alternative *short-form* syntax exists. This short-form syntax still requires matrix contents be enclosed by square brackets `[ ... ]`, but allows one to substitute *spaces* to delimit columns and *newlines* to delimit rows. For example, enter the command:

```
X = [
1 2 3
4 5 6
7 8 9
]
```

MATLAB will output the expected result:

```
X =

    1    2    3
    4    5    6
    7    8    9
```

Note: Technically, there are additional *newlines* after the initial '`[`' and before the closing '`]`'. These newlines do not delimit a row—they are simply syntactic sugar.[8]

It is also possible to mix and match the short-hand and standard syntax for defining a matrix:

```
X = [
1 2, 3
4, 5 6;
7, 8 9
]
```

However, the most common form of mixing and matching is to use commas (`,`) to delimit columns and *newlines* to delimit rows. For example:

```
X = [
1, 2, 3
4, 5, 6
7, 8, 9
]
```

The reason that commas are preferred to *spaces* is that they are easier to read for matrix definitions that contain mathematical expressions. Consider how difficult this is to read:

```
X = [
2 + exp(0.1) 35
-70 -1 + tan(1.27)
]
```

It is unclear that -70 is a separate expression from the -1 that follows.

As a general rule, use commas whenever your matrix contains mathematical expressions. Both semicolons (`;`) and *newlines* see roughly equal use as the delimiter of rows.

1.5.4 Generating Vectors and Matrices

So far, we've looked at initializing vectors/matrices by entering each element individually. This can become inconvenient for vectors/matrices containing only a single (repeated) value. Consider the 3×7 matrix:

$$\mathbf{A} = \begin{bmatrix} 3, & 3, & 3, & 3, & 3, & 3, & 3 \\ 3, & 3, & 3, & 3, & 3, & 3, & 3 \\ 3, & 3, & 3, & 3, & 3, & 3, & 3 \end{bmatrix}, \tag{1.6}$$

which consists entirely of *threes*—or the vector 100-length vector:

$$\mathbf{b} = \underbrace{\left[1,\ \ldots,\ 1\right]}_{\text{100 entries}}, \tag{1.7}$$

consisting entirely of *ones*. Also consider vectors that are sequences of consecutive numbers such as:

$$\mathbf{c} = [1,\ 2,\ \ldots,\ 99,\ 100]. \tag{1.8}$$

It is impractical to enter the elements of these vectors/matrices manually. Luckily, MATLAB has many built-in functions capable of generating common vectors and matrices.

Generating Vectors of a Single Value

To generate vectors/matrices consisting of a single (repeated) value, MATLAB offers two convenient functions: `zeros` and `ones`. Try using the `help` command for both functions to see how they are called.

As an example, to create a 3×2 matrix of zeroes, enter the command:

```
1  A = zeros(3,2)
```

MATLAB yields the output

```
1  A =
2
3       0     0
4       0     0
5       0     0
```

To create a 2×4 matrix of ones, enter the command:

```
B = ones(2,4)
```

MATLAB yields the output

```
B =

     1     1     1     1
     1     1     1     1
```

Note that both the `zeros` and `ones` commands generate matrices by default. In other words, calling `ones(3)` does not produce a 3×1 vector! In your lab report, comment on the output of running the `ones(3)` command.

To generate a matrix with arbitrary repeated values, one can multiply the desired value (a scalar) to the `ones` command. For example, enter the command:

```
C = 5*ones(3,3)
```

MATLAB yields the output

```
C =

     5     5     5
     5     5     5
     5     5     5
```

In a way, this makes the `zeros` command redundant (as one could achieve a similar effect using `0*ones(...)`). This multiplication syntax leverages the interactions between matrices and scalars (detailed later in Sec. 1.5.5).

In your lab report, give the MATLAB command to generate $\mathbf{A}$ and $\mathbf{b}$ from Eqs. (1.6) and (1.7).

Generating Sequential Vectors

In addition to generating matrices of a single (repeated) value, MATLAB also provides built-in mechanisms for generating sequential vectors. The most common mechanism for generating is via the *colon operator*.[9]

In its most general form, expressions involving the *colon operator* have the syntax:

$$\mathtt{x} = j : i : k \tag{1.9}$$

where j is the *initial value*, i is the *step size* (or *increment*), and k is the *final value*. This will generate a vector from j to k at evenly spaced intervals i. For example, enter the following command:

```
x = 1:1:5
```

MATLAB yields the output

```
x =

    1     2     3     4     5
```

In the scenarios where the *step size/increment* is *one*, the i in the expression can be omitted as in the following example; enter the following command:

```
x = 1:5
```

Does this yield the same result as `1:1:5`? Comment on this in your report.

In instances where the step size is *not* one, the i parameter is required. For example, consider the case where $i = 2$; enter the following command:

```
x = 1:2:5
```

MATLAB yields the output

```
x =

    1     3     5
```

In this example, i is set to a fractional amount; enter the command:

```
x = -1:0.5:1
```

It is also possible to have *step sizes/increments* that are negative. In this case, one should ensure that $k < j$. For example, enter the following command:

```
x = 5:-1:1
```

Another common scenario that occurs is one in which you wish to generate sequential vectors for which one knows the *initial value*, the *final value*, and the *total number of elements* in the vector. While it's possible to calculate the required *step size/increment* from these data points, MATLAB provides a built-in function, `linspace`, which generates vectors using this set of parameters. Use the `help` command to read the documentation for `linspace`:

```
help linspace
```

Now for a few examples. To generate a sequential vector from 1 to 2 containing a total of 5 elements, enter the command:

```
x = linspace(1,2,5)
```

MATLAB yields the output

```
x =

    1.0000    1.2500    1.5000    1.7500    2.0000
```

Often, an odd number of total elements are preferred. To see this, enter the command:

```
x = linspace(1,2,4)
```

MATLAB yields the output

```
x =

    1.0000    1.3333    1.6667    2.0000
```

Here, we can see that an even total number of elements means that the midpoint between the *initial* and *final* values is not contained within the sequence—this is usually not preferred. In

your lab report, give the MATLAB command that will generate **c** from Eq. (1.8).

1.5.5 Operations on Vectors and Matrices

Care must be taken when performing arithmetic operations with vectors/matrices. This is because MATLAB will attempt to conform to the linear algebra definition of mathematical operation. This can occasionally lead to results you might not expect. Consequentially, we will discuss the various arithmetic operations between combinations of vectors/matrices and scalars.

Operations Between Vectors/Matrices and Scalars

Let's detail the arithmetic operations between vectors/matrices and scalars. Our examples will consist of an example vector **a** and the scalar 3. We will start with addition (+), enter the commands:

```
a = 1:4;
b = a + 3
```

MATLAB returns the output:

```
b =

     4     5     6     7
```

Moving to subtraction (−), enter the command:

```
b = 3 - a
```

MATLAB returns the output:

```
b =

     2     1     0    -1
```

Multiplication (∗) works as expected. Enter the command:

```
b = 3 * a
```

MATLAB returns the output:

```
b =

     3     6     9    12
```

Division (/) does not work as expected. Enter the command:

```
b = 3 / a
```

What does MATLAB return? Comment on this result in your report.

All of the operations so far: addition (+), subtraction (−), and multiplication (∗) between a scalar and a matrix/vector have had a technical, mathematical definition in linear algebra. Unfortunately, dividing (/) a scalar by a vector/matrix *does not.* However, dividing (/) a vector/matrix by a scalar *is defined,* and, consequently, the following command works properly—enter the command:

```
b = a / 3
```

MATLAB returns the following result:

```
b =

    0.3333    0.6667    1.0000    1.3333
```

Returning to the matter of dividing a scalar by a vector—though there may not be a linear algebra definition, a desirable behavior might be:

$$\mathbf{a} = \begin{bmatrix}1, & 2, & 3, & 4\end{bmatrix}$$
$$3/\mathbf{a} = \begin{bmatrix}3/1, & 3/2, & 3/3, & 3/4\end{bmatrix}.$$

This behavior is called *element-wise division* and is defined in MATLAB by the *period divide* (`./`) operator.[10] Enter the command:

```
b = 3 ./ a
```

Does MATLAB output the expected result? Comment on the result in your report.

Like division, taking a vector/matrix to a scalar power will not operate as you may initially

expect. Enter the command:

```
b = a ^ 3
```

What result does MATLAB produce? This result occurs because MATLAB is attempting to perform the exponentiation as three consecutive matrix products, i.e., $\mathbf{a} \cdot \mathbf{a} \cdot \mathbf{a}$ which is not possible to evaluate for a row vector.

Instead, a desirable behavior for taking a vector/matrix to a scalar power is:

$$\begin{aligned} \mathbf{a} &= \begin{bmatrix} 1, & 2, & 3, & 4 \end{bmatrix} \\ \mathbf{a}^3 &= \begin{bmatrix} 1^3, & 2^3, & 3^3, & 4^3 \end{bmatrix}. \end{aligned}$$

This operation is defined in MATLAB as *element-wise power* and is defined using the *period caret* (`.^`) operator.[11] Enter the command:

```
b = a .^ 3
```

Does it behave as expected? Comment on the result in your report.

This choice of behavior may seem frustrating at first, but MATLAB was designed around linear algebra and will always attempt to implement the rigorous definition of linear algebra operations unless explicitly told not to. If you continue to use MATLAB throughout your career, you may begin to appreciate why this choice was made as you grow your experience writing MATLAB code.

1.5.6 Operations Between Vectors/Matrices

Just as arithmetic operations are defined between a vector/matrix and a scalar, so too are operations between two vectors/matrices. In the case of addition and subtraction, there is the added restriction that the two vectors/matrices have the same *length*/*size*. For example, type the following commands:

```
a = 2:4;
b = 1:3;
c = a + b
```

MATLAB will produce the output:

```
c =

     3     5     7
```

It is also possible to subtract `b` from `a`.

What happens if you try to add/subtract vectors of different lengths? Enter the commands:

```
d = 1:4;
c = a + d
```

Comment on the result in your lab report.

Attempting to multiply `a` and `b` using the command: `c = a * b`, will result in an error as MATLAB will attempt to evaluate the matrix product between `a` and `b`—which isn't possible for two vectors of the same orientation.

Still, it is desirable to multiply two vectors:

$$\begin{aligned} \texttt{a} &= \begin{bmatrix} 2, & 3, & 4 \end{bmatrix} \\ \texttt{b} &= \begin{bmatrix} 1, & 2, & 3 \end{bmatrix} \\ \text{such that} \quad \texttt{a * b} &= \begin{bmatrix} 1 \cdot 2, & 2 \cdot 3, & 3 \cdot 4 \end{bmatrix}. \end{aligned}$$

This is accomplished by using *element-wise multiplication*. Enter the command:

```
c = a.*b
```

Confirm that the result is as you would expect. Given your knowledge so far, explain how the *element-wise division* (`./`) and *element-wise power* (`.^`) behave when applied `a` and `b`.

A safe rule of thumb for these lab exercises is to use (`.*`) and (`./`) as you will rarely need to use their linear algebra-compliant companions.

1.5.7 Functions applied to Vectors and Matrices

Now that we've defined arithmetic operations for vectors/matrices, we will turn our attention to applying functions to vectors/matrices (in other words, passing a vector/matrix as a function

parameter). We will begin by exploring functions that require vectors as input. The first class of functions we will visit are the *sizing* functions. These consist of: `numel`, `size`, and `length`. Enter the command:

```
A = ones(5,4);
```

The `numel` function will return the total number of elements in `A`. Enter the command:

```
numel(A)
```

MATLAB produces the output:

```
ans =

   20
```

as expected ($5 \cdot 4 = 20$).

The `size` function returns the *size* (or *dimensionality*) *as a vector*. Enter the command:

```
size(A)
```

MATLAB produces the output:

```
ans =

     5     4
```

The `length` command returns the largest dimension. Enter the command:

```
length(A)
```

MATLAB produces the output:

```
ans =

     5
```

For vectors, `numel` and `length` will produce equivalent results. In your lab report, explain why this is the case.

Other classes of functions *select* values from a vector/matrix passed to it; the `min` and `max` functions will return the minimum or maximum value contained with the vector passed as a

parameter. For example: `max([10, 2, 109, 50])` returns 109.

The final class of functions are functions that behave as *maps* when passed a vector. Mapping[12] is a powerful concept that is a foundation pillar of the *functional programming paradigm*.[13] The idea of a *map* is that, for a function, $f(x)$ which is defined for x as a scalar, the extension to $f(\mathbf{x})$ for an n-length vector $\mathbf{x}$ is:

$$f(\mathbf{x}) = \left[f(x_1), \quad f(x_2), \quad \ldots, \quad f(x_{n-1}), \quad f(x_n) \right]$$

In layman's terms, a function (defined for scalars) is extended to take vectors/matrices by applying the function to each element within the vector. Functional mapping enables MATLAB to avoid using explicit *for loops* and delegate much of this responsibility to functional maps instead. Some of the functions which behave as maps are:

- the trigonometric functions `sin`, `cos`, and `tan`,
- the inverse trigonometric functions `asin`, `acos`, and `atan`,
- the exponential function `exp`, and
- the logarithm functions `log` and `log10`.

Whenever you encounter a new function and are curious if it can be used as a map, you can always use the `help` command or search the MathWorks website[6] for documentation.

1.6 Basic Plotting

Now that we've covered how to generate vectors and how vector math works in MATLAB, it's time to start doing something more interesting—plotting! Supposed we are asked to plot the signal:

$$x(t) = \sin(2\pi t/4) \quad \text{for} \quad 0 \leq t \leq 4.$$

Here $x(t)$ is a function of t and we are asked to plot it over $0 \leq t \leq 4$. The first thing we need to do is to create a time vector t. Enter the command:

```
t = linspace(0, 4, 501);
```

Here, we've chosen `t` as a vector spanning the range from 0 to 4 containing 501 intermediate points. The number 501 is chosen somewhat arbitrarily. We want a sufficiently large number to provide a plot with reasonable resolution: 500 should be sufficiently large. We opt for 501 as using an odd number of intermediate points is generally good practice.

Now let's compute $x(t)$ given our time vector `t`. Because `sin` is a *mapping function* (as mentioned in Sec. 1.5.7), we can pass a vector (in this case, a scaled version of `t`: $2\pi\, t/4$) as a parameter to `sin` and it will return a vector containing the values of $x(t)$ for each time in vector `t`. Enter the command:

```
1 xt = sin(2*pi*t./4);
```

Finally, to plot `xt` with respect to `t`, enter the command:

```
1 plot(t, xt)
```

MATLAB should produce the graphical out given in Fig. 1.3.

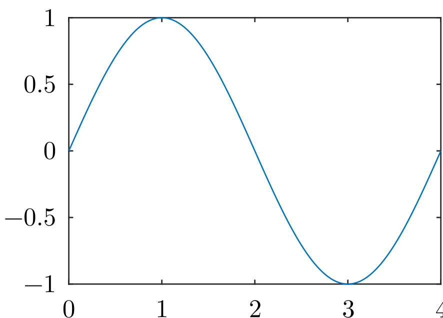


Figure 1.3: Graphical output produced by MATLAB.

Use `help` on the `plot` function to read about how it works. In our case, the first parameter corresponds to a vector containing the values for the x-axis, and the second parameter corresponds to a vector containing the values for the y-axis. It is also possible to call the `plot` function with only *one* parameter. Enter the command:

```
1 plot(xt)
```

MATLAB should produce a slightly different graphic. In your lab report, comment on the difference in graphical output. Why do you suppose this is?

Return to the version of `plot` which uses both parameters; enter the command:

```
plot(t, xt)
```

We can add labels and a title to the currently opened figure with the following commands; enter the following commands:

```
title('Sinusoid')
xlabel('t')
ylabel('x(t)')
```

MATLAB should produce the graphical out given in Fig. 1.4. Don't close the window.

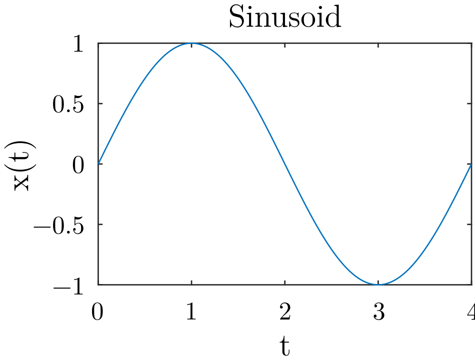


Figure 1.4: Graphical output produced by MATLAB, now with title and labels.

Notice that the values passed to the `title`, `xlabel`, and `ylabel` commands are enclosed by single quotes (`'`). These single quotes tell MATLAB that their contents constitute a *character array*[14] (this is roughly analogous to *strings* in other programming languages). This tells MATLAB to treat the contents of these commands as *plain text* rather than attempting to parse them as MATLAB commands.

You may skip this paragraph if you are *not* already familiar with strings. For those familiar with strings in other programming languages, you'll note that many other languages encode strings using double quotes (`"`). MATLAB also has a datatype called *strings* which are instantiated using double quotes (`"`). However, you should *avoid* them (at least initially) as they require knowledge of advanced topics like *cells arrays*. A few core MATLAB functions also require their parameters to be *character arrays* and will reject *strings*. It might be hard to kick the habit of using double quotes (`"`) to denote strings, but be advised that doing so will eliminate some errors that might pop up otherwise. Yes—it's strange/frustrating.

1.6.1 Changing the Viewing Region

Look at the graph in Fig. 1.4. Note that the peaks of the sinusoid touch the edges of the *viewing region*. It is often desirable to modify this viewing region. In our case, it would be nice to display the region between ± 1.5 on the vertical axis (rather than the ± 1 it is currently set to). This way, the peaks of the sinusoid will no longer graze the edge of the viewing window.

MATLAB provides the `axis` function to change the viewing region of the current graph. This command lets you define the minimum/maximum values visible for the x- and y-axis. This command does not take four parameters to specify the view region. Rather, it takes a **single parameter**, a row vector containing four elements. The row vector has the form:

$$\left[x_{\text{min}}, \quad x_{\text{max}}, \quad y_{\text{min}}, \quad y_{\text{max}}. \right]$$

To change the view region so that the y-axis spans from -1.5 to 1.5, we first note that we want to keep the x-axis as it is. From Fig. 1.4 we see that the x-axis ranges from 0 to 4. The following command will adjust the viewing region as we desire. Enter the command:

```
axis([0, 4, -1.5, 1.5])
```

MATLAB should produce the graphical output given in Fig. 1.5.

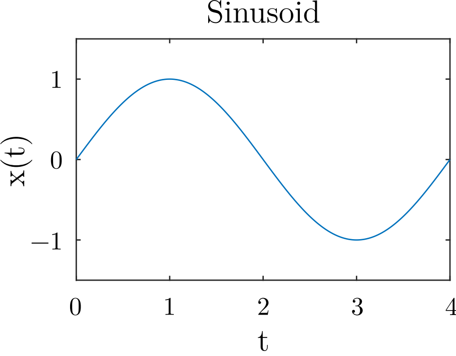


Figure 1.5: Graphical output produced by MATLAB with a more comfortable view.

As an exercise to the reader, provide the MATLAB command that produces the graphical output depicted in Fig. 1.6. Include this in your lab report.

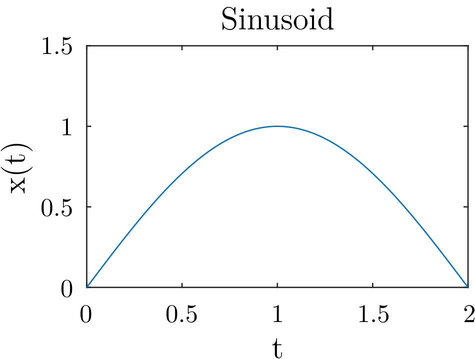


Figure 1.6: Graphical output produced by MATLAB—alternate view.

1.6.2 Plot-like Functions

For you to truly develop an understanding of what the plot function is actually doing, we are going to reduce the number of points in the time vector from 501 to 5 and re-initialize `xt` based on this new time vector. Enter the commands:

```
t = linspace(0, 4, 5);
xt = sin(2*pi*t./4);
plot(t, xt)
```

MATLAB should produce the graphical output given in Fig. 1.7.

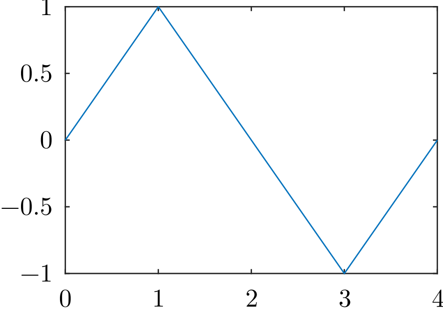


Figure 1.7: Graphical output produced by MATLAB—now jagged.

Why is this graph jagged? Isn't the function $x(t)$ supposed to be a sinusoid (which is smooth)? Explain what is happening in your lab report.

Whatever your explanation was for the previous question, one thing is for sure, both our time vector `t` and `xt` have a small number of elements (5 to be precise). Whenever we have

have small (with respect to the number of elements), it is sometimes useful to use an alternative plotting function. Functions that produce *plots* are called *plot-like functions*.[15] These functions generally share a similar syntax to `plot`. One such function is `stem`. Enter the command:

```
stem(t, xt)
```

MATLAB should produce the graphical output depicted in Fig. 1.8.

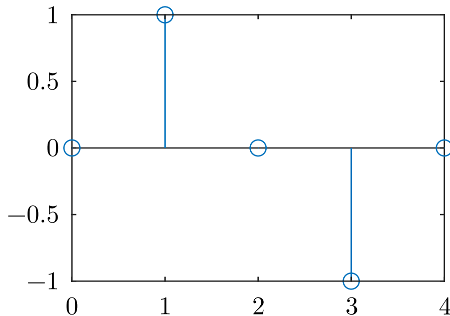


Figure 1.8: Graphical output produced by MATLAB.

The `stem` function is often used to plot signals we wish to denote as discrete time. This function only works well for short (with respect to the number of elements) signals. To see why, run the following commands. Enter the commands:

```
t = linspace(0, 4, 501);
xt = sin(2*pi*t./4);
stem(t, xt)
```

Comment on the graphic produced in your lab report.

Another common *plot-like* function is `scatter`. Enter the command:

```
scatter(t, xt)
```

Comment on the type of graphic that the `scatter` function produces in your lab report.

1.7 Utilizing Scripts

Until now, we've entered commands directly into the MATLAB **Command Window**. The command window is a great way to run a few commands and get some immediate feedback.

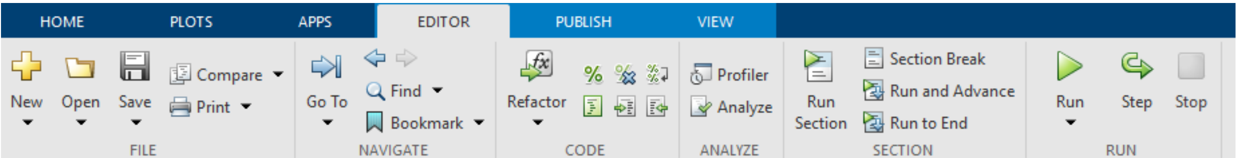


Figure 1.9: The Upper Menu with the Editor tab selected.

However, the **Command Window** is an inconvenient method for tasks that require more than a few commands (like plotting signals or completing future lab excercises). Fortunately, MATLAB has the concept of *scripts*. A *script* is a *collection of commands* executed together and can be saved and loaded at a later time.

Most of your work from this point forward will take place within a script rather than the **Command Window**. Running scripts is generally the preferred method of executing MATLAB commands. Scripts are edited in the **Script Editor Window**. You should see an empty script in that window. If you do not, press the keyboard shortcut `Ctrl+N` to create a new script.

Select the **Script Editor Window** and ensure the script is empty. Retype the following commands to plot the sinusoid graph from Fig. 1.4:

Listing 1.1: `plot_sinusoid.m`

```
t = linspace(0, 4, 501);
xt = sin(2*pi*t./4);
plot(t, xt)
title('Sinusoid')
xlabel('t')
ylabel('x(t)')
```

Ensure that the **Editor** tab of the upper menu is selected—as in Fig. 1.9. Click the **Save Icon** and save the script as `plot_sinusoid.m`. Ensure that the directory in the **Address Bar** in MATLAB (see Fig. 1.1) matches the directory you saved your script. If this is the case, you should see the `plot_sinusoid.m` script in MATLAB's **Current Folder View**—as in Fig. 1.10.

It should be mentioned that when picking names for scripts, one should make certain not to choose a name which *shadows* a built-in MATLAB function! This means you should *not* name your scripts things like `plot.m`, `exp.m`, or `linspace.m` etc. Doing so will confuse MATLAB and cause it to attempt to call your script instead of the MATLAB built-in.

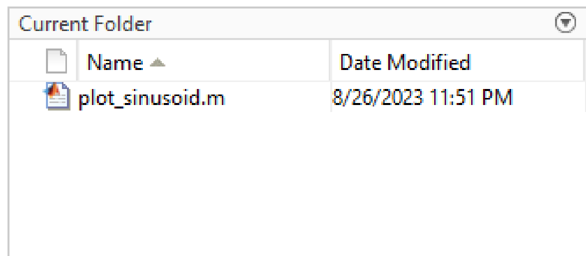


Figure 1.10: The Current Folder View.

With that out of the way, try *running* the script. Do so by pressing the **Run** button in the **Upper Menu**. You should see the same graph as depicted in Fig. 1.4.

Now let's learn about *comments*. In MATLAB, comments are specified by the percent (%) symbol. Everything to the right of the percent (%) symbol until the end of the line is considered a *comment* and is not executed by the interpreter. Comments are a helpful way to document *what/why* your code is doing.

For instance, we could add some comments to our `plot_sinusoid.m` example:

Listing 1.2: `plot_sinusoid.m` (*commented*)

```
t = linspace(0, 4, 501); % generate the time vector
xt = sin(2*pi*t./4); % define the signal vector

% plot the Signal
plot(t, xt)
title('Sinusoid')
xlabel('t')
ylabel('x(t)')
```

There is an important property of scripts that should be mentioned before moving much further. This property is somewhat unique to MATLAB if you come from languages like C or Java. This property is best demonstrated through an example. Remove line 1 from Listing 1.2.

This now removes the definition of the time-vector, `t`. The line immediately below it, where `xt` is defined, depends on `t`. One would imagine that running the script now would throw an error. Wrong! Run the script again!

The script runs just fine! This is because MATLAB retains *program state* between script calls. This means that MATLAB remembers the definition of `t` from the previous time the script was

run. In fact, you can see this definition of `t` in the **Variable Workspace**. Remember, a script is simply a series of commands that get run sequentially—and since MATLAB retains *program state* between commands (otherwise, you could never define a variable in the command line), MATLAB must also retain state between script calls. This is a source for frustrating and difficult-to-debug bugs. As a consequence, **it is best practice to insert the following line at the beginning of each script:**

```
clearvars; close all; clc;
```

The `clearvars` command clears all the variable definitions. The `close all` command closes all of the plots and figures (trust me, you want this—it prevents a lot of frustrating bugs that crop up later). Finally, `clc` clears the command line printout. This ensures that only output your script produces is visible in the command line—aiding in debugging.

1.8 Advanced Plotting

Until now, we've been producing graphical content using `plot` or the other *plot-like* functions. But there is *a lot* that has been happening under the hood. In this section, we will be introduced to the *three* main components of graphical output: *figures*, *axes*, and *plots*.

Actually, we've already used all three of these components—the process for instantiating them was handled for us via the `plot` command. Now, we will conduct our exploration by invoking them manually.

1.8.1 Multiple Figures

When MATLAB produces graphical output, the windows that pop up are called *figures*. Create a new script called `multiple_figures.m` and add the following contents:

Listing 1.3: `multiple_figures.m`

```
clearvars; close all; clc;
t = linspace(0,2,501); % time vector
x = 0.7*exp(-4.2*t).*sin(6.4*2*pi*t); % an example function x(t)
y = exp(-2.5*t).*cos(1.4*2*pi*t); % an example function y(t)
```

```
5
6 figure(1)
```

Run the script. A blank window should appear; this is a *figure*. The function `figure(1)` creates a new figure with id = 1 and sets the *current figure* to 1. In the case where a figure with id = 1 *already exists*, the command `figure(1)` will *refocus* the *current figure* by setting its value to 1. What is the *current figure*? MATLAB maintains a reference to the current working figure. This is the figure that all subsequent graphical operations will apply to.

An *axes* (not to be confused with *axis*) is a region within a *figure* that `plots` can be added to. Add the following line to your `multiple_figures.m` script:

```
7 axes()
```

Run the script. You should now see an empty region/grid on the figure; this is an *axes*. The `axes()` function adds an *axes* to the *current figure* and sets the *current axes* to the *axes* created by the `axes` function (unlike *figures*, *axes* do not have id numbers).

Now add a *plot* to the *current axis* using the `plot` command. A *plot* is anything produced by a *plot-like* function. In the case of `plot`, this is the blue curve. Add the following line to your `multiple_figures.m` script:

```
8 plot(t,x);
```

Run the script. Comment on what you see in your lab report.

Given a *plot* merely refers to the blue curve we've seen in past sections, how was it that the `plot` function was able to produce any graphical output before if we hadn't created a *figure* and *axes* to plot *to*? Well, in the instance that `plot` is called and there is no *current figure* and no *current axes*, the `plot` function (or any *plot-like* function) will create a new *figure* and *axes* for it to plot onto.

Now let's make our `multiple_figures.m` script live up to its name. Add the following lines to your `multiple_figures.m` script:

```
9 figure(2)
10 plot(t,y);
```

Run the script. Comment what you see in your lab report. Note that we didn't explicitly add

an *axes* command for the second *figure*; this was handled by `plot(t,y)`.

Your final exercise is to modify the `multiple_figures.m` script to add titles and label the x-axis for both figures. The title for the first figure should be "x(t)" and the title for the second figure should be "y(t)." The x-axis label for both figures should be "t." *Hint:* The `title` and `xlabel` functions apply their result to the *current axes* of the *current figure*.

1.8.2 Subplots

It is possible to have multiple *axes* on a single *figure*. This is done using the `subplot` function. Despite its name, this function is *not* a *plot-like* function—instead this function creates axes on the *current figure*. The `subplot` function carves up the *current figure* in a grid structure, places an *axes* in that grid, and sets the *current axes* to that *axes*. In the **Command Window**, use the `help` command on `subplot` to read its documentation.

The `subplot` command takes three parameters: the total number of rows (m) to divide the figure into, the total number of columns to divide the figure into (n), and the *subplot index* (p) of the *axes* within the grid to set as the *current axes*. These p indices are assigned beginning at the top-left of the grid and increase sequentially—first by column, then by row.

This is more easily explained through an example. Suppose you wanted to create *four* axes arranged in a 2×2 grid on the *current figure*. You would create the top-left *axes* with the command `subplot(2,2,1)`. The first `2` indicates that there are two total rows, the second `2` indicates that there are two total columns and 1 indicates that we are selecting the top-left *axes* from that grid. To select the top-right axes, we would call `subplot(2,2,2)`. To select the bottom-left axis, we would call `subplot(2,2,3)`, and to select the bottom-right, we would call `subplot(2,2,4)`. This entire configuration is depicted in Fig. 1.11.

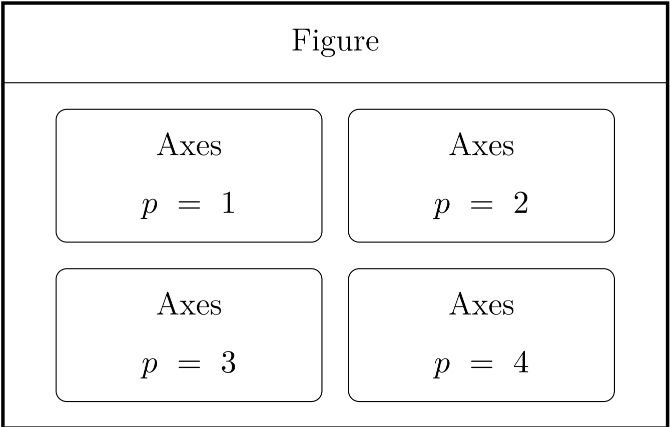


Figure 1.11: Illustration of how p indices are assigned for 2×2 *axes* within a *figure*.

For our example, we will create two *axes* arranged in a row. Create a new script called `subplots_example.m` and add the following contents:

Listing 1.4: `subplots_example.m`

```
clearvars; close all; clc;
t = linspace(0,2,501); % time vector
x = 0.7*exp(-4.2*t).*sin(6.4*2*pi*t); % an example function x(t)
y = exp(-2.5*t).*cos(1.4*2*pi*t); % an example function y(t)

figure(1);

% two axes in a row means we have a total of 1 row, 2 columns
% left axes
subplot(1, 2, 1) % 1 row, 2 columns, p=1
plot(t, x)
title('x(t)')
xlabel('t')

% right axes
subplot(1, 2, 2) % 1 row, 2 columns, p=2
plot(t, y)
title('y(t)')
xlabel('t')
```

Run the script. MATLAB should produce the graphical output depicted in Fig. 1.12.

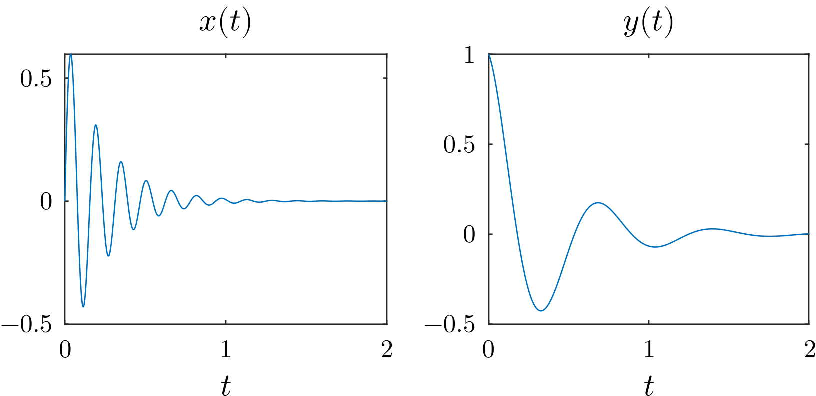


Figure 1.12: Graphical output for two *axes*/*subplots* arranged in a row.

As an exercise to the reader, modify the `subplot` commands in your `subplots_example.m` script so that the *axes* are arranged in a column, rather than in a row. Include this modified script within your report.

1.8.3 Multiple Plots on the Same Axes

Finally, it is possible to output multiple *plots* on the same *axes*. Normally, *plot-like* functions overwrite any previously plotted data on the *axes*. For example, create the script `multiple_plots.m` and add the following contents:

Listing 1.5: `multiple_plots.m`

```
clearvars; close all; clc;
t = linspace(0,2,501); % time vector
x = 0.7*exp(-4.2*t).*sin(6.4*2*pi*t); % an example function x(t)
y = exp(-2.5*t).*cos(1.4*2*pi*t); % an example function y(t)

figure(1);
plot(t, x)
plot(t, y) % this plot function will overwrite the previous function
```

Run the script. You should see only one plot.

We can change this behavior by updating the *hold* property of the *current axes*. By default,

the *hold* property is *off*. The `hold on` command sets the *current axes hold* property to *on*. Modify your `multiple_plots.m` script so that it reads:

Listing 1.6: `multiple_plots.m`

```
clearvars; close all; clc;
t = linspace(0,2,501); % time vector
x = 0.7*exp(-4.2*t).*sin(6.4*2*pi*t); % an example function x(t)
y = exp(-2.5*t).*cos(1.4*2*pi*t); % an example function y(t)

figure(1);
plot(t, x)
hold on;
plot(t, y) % this plot function will overwrite the previous function
legend('x(t)', 'y(t)')
```

Notice that the `hold on` command has been added after `plot(t, x)`. The legend command tells MATLAB to add a legend to the *current axes*: labeling the first plot as "x(t)" and the second plot as "y(t)." Run the script. MATLAB should produce the graphical output depicted in Fig. 1.13.

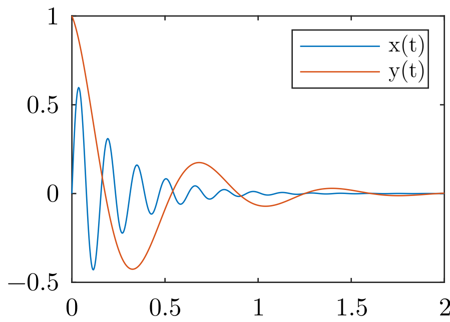


Figure 1.13: Two *plots* on the same *axes*.

It is possible to overlay any *plot-like* command. As an exercise to the reader, change the second `plot` function to `scatter`. Comment on the result in your lab report.

Now, we will demonstrate the importance of the `clearvars; close all; clc;` line in scripts. First, re-run `multiple_plots.m` (if needed) so that the previous graphical output is visible. Don't close the window. Next, modify two lines of `multiple_plots.m`. First, remove the first

line (`clearvars; close all; clc;`). Next, make a slight change to either the definition of `x` or `y` (for instance, change `x` to `x = 0.3*exp(-4.2*t).*sin(6.4*2*pi*t);`). Now, re-run the script. Comment on what happens in your lab report.

This occurs because MATLAB retains *program state*—because you didn't close all the figures at the start of the script. MATLAB remembers that the *axes hold* property is set to *on*. When the script is re-run, MATLAB re-uses the same *figure* and *axes* from the previous run. Because the *hold* is *on*, when MATLAB encounters `plot(t,x)` after the script has been run once, it won't overwrite the `plot` data from the previous run.

1.8.4 Plotting Complex-valued Functions

Let's combine all the plotting concepts we've learned so far. Consider the complex-valued function,

$$z(t) = e^{-t+\mathrm{j}2\pi t} \quad \text{for} \quad 0 \leq t \leq 5. \tag{1.10}$$

Plotting complex functions presents a challenge as the values they return contain two separate components: real and imaginary (or magnitude and phase). There are two common methods for plotting complex functions:

1. plotting each component separately, or
2. parametric plots.

We will explore both methods for plotting $z(t)$.

Create a new script: `complex_exponential.m` and add the standard top line:

Listing 1.7: `complex_exponential.m`

```
1 clearvars; clc; close all;
```

From Eq. (1.10) we should first define our time vector, t as:

```
2 t = linspace(0, 5, 501);
```

Now let's define $z(t)$ based on the definition from Eq. (1.10):

```
3 zt = exp(-t + 2j*pi*t);
```

Next, we will extract the real and imaginary components of $z(t)$ of with the code below:

```
re_zt = real(zt);
im_zt = imag(zt);
```

Now plot the real and imaginary components of $z(t)$ over time on the same plot.

```
figure(1);
plot(t, re_zt);
hold on;
plot(t, im_zt);
xlabel('t');
legend('Re(z(t))', 'Im(z(t))');
```

Comment on what you see.

Next we are going to plot $z(t)$ in the complex plane (a *parametric* plot):

```
figure(2);
plot(re_zt, im_zt);
xlabel('Re(z(t))');
ylabel('Im(z(t))');
title('z(t)=exp(t + j2{\pi}t)')
```

Note: the portion of the title string: `{\pi}`, tells MATLAB to print the symbol, π.

Comment on what you see. Explain it in terms of the behavior of the real and complex exponentials and sinusoids. Be sure to explain why we see a spiral instead of a circle.

1.9 Post-Lab Questions

Q.1

Suppose `x` is a vector. Which command is used to find the *number of elements* in `x`?

Q.2

Suppose `X` is a matrix. Which command is used to find the *size/dimensions* of `X`?

Q.3

- Suppose a student constructs a vector: `x = 0:3`.
  - Write out the contents of `x`.

  - How many (*total*) elements are there in `x`?

- Suppose a student constructs a vector: `y = 0:100`. How many (*total*) elements are there in `y`?
- In general, for a positive integer n, how many (*total*) elements will be in a vector `z = 0:n`?

Q.4

A student types the following command: `subplot(3, 4, 5);`

Which particular *axes* (row/column) does this select?

Q.5

A student makes a new script and names it `plot.m`. What's wrong?

Q.6

A student defines two vectors:

```
a = [1, 2, 3];
b = [4, 5, 6];
```

The student wishes to multiply `a` and `b` together to produce the result `[4, 10, 18]`.

The student types:

```
c = a * b;
```

Does `c` contain the result the student expects? How can the student obtain the desired result?

Q.7

In MATLAB, what is the difference between an *axis* and an *axes*?

Q.8

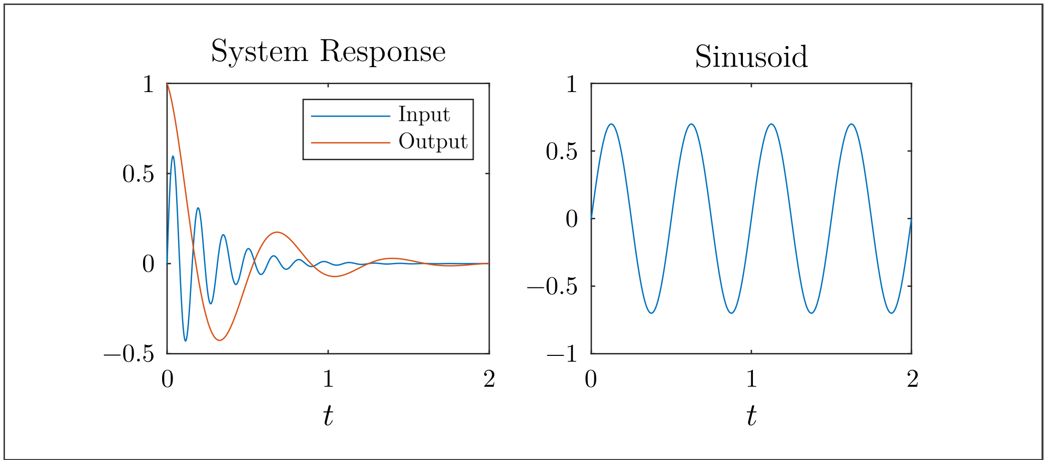


Figure 1.14: Graphical output produced by MATLAB. The entire output is contained within a single window.

Given that MATLAB graphical output depicted in Fig. 1.14 occurs *within a singular window*,

- How many total *figure*(s) are present?
- How many total *axes*(es) are present?
- How many total *plot*(s) are present?

Q.9

A student writes the following script, `dual_sinusoids.m`, to plot 2 sinusoids:

Listing 1.8: `dual_sinusoids.m`

```
t = linspace(0,2,501);
x = 0.2*cos(2*2*pi*t+0.72);
y = 0.6*cos(3*2*pi*t-0.9);

figure(1);
plot(t, x);
hold on;
plot(t, y);
```

The student runs their script and realizes they need to update the definition of `x`. The student updates `x` to `x = 0.8*cos(2*2*pi*t+0.72)` and reruns their script.

However, rather than plotting two sinusoids—*something unintended happens.* What

unintended behavior occurred? What can the student add to their script to make it function as intended?

Q.10

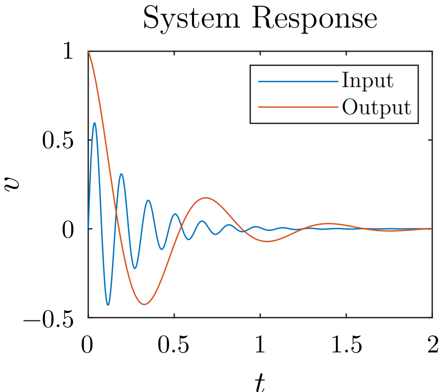


Figure 1.15: The MATLAB *axes* the student wishes to enlarge.

A student produces the *axes* given by Fig. 1.15, but they wish to enlarge the *axes* such that it depicts the region between $1 \leq t \leq 2$ and $-0.5 \leq v \leq 0.5$. Which command can they use to accomplish this?

1.10 Report Checklist

Work with lab partner(s) to create a lab report for this lab exercise. Be sure to include the names of each person in your group on the report. This report should be a typed document that outlines the basic procedure of the lab exercise and should include any results you were asked to obtain. Your report should follow a *cohesive narrative*—it should always be apparent *why* you have presented the results you have. Do not simply insert results without (1) motivating the problem and (2) providing an analysis of the data. Ensure that your report is grammatically correct with complete sentences. This does not mean your report must be wordy with filler—in fact, avoid filler and state your ideas concisely.

The final Post-Lab Questions section does need to be woven into your lab narrative. You may simply append your answers to each question at the end of your report in an attached answers

section. Be sure to indicate which question you are answering by referencing the corresponding question number.

Be sure the following are included in your report:

- Section 1.3: Comment on what happens when the command `z = y + 3` is run immediately after running `clearvars`.
- Section 1.4.1: Evaluate the expression. Provide both MATLAB code and result.
- Section 1.4.4: Comment on the output of a complex number defined in polar form. Does MATLAB print the result in rectangular or polar form?
- Section 1.5.2: Give the *size* (or *dimensionality*) of the matrix $\mathbf{Y}$ from Eq. (1.5).
- Section 1.5.3: Give MATLAB commands to initialize $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{Y}$ from Eqs. (1.2)–(1.3), (1.5).
- Section 1.5.4: Comment on the output of `ones(3)`. Is it a vector or a matrix?
- Section 1.5.4: Give the MATLAB command to generate $\mathbf{A}$ and $\mathbf{b}$ from Eqs. (1.6)–(1.7).
- Section 1.5.4: Comment on whether `1:1:5` is equivalent to `1:5`. Why?
- Section 1.5.4: Give the MATLAB to generate $\mathbf{c}$ from Eq. (1.8).
- Section 1.5.5: Comment on MATLAB's output when one attempts to divide a scalar by a vector/matrix.
- Section 1.5.5: Comment on the result of using *element-wise division* (`./`).
- Section 1.5.5: Comment on the result of applying *element-wise power* (`.^`).
- Section 1.5.6: Comment on the result of adding two vectors of different lengths.
- Section 1.5.6: Explain how the *element-wise division* (`./`) and *element-wise power* (`.^`) behave when applied to vectors $\mathbf{a}$ and $\mathbf{b}$.
- Section 1.5.7: Explain why `numel` and `length` produce the same results when passed a *vector*.
- Section 1.6: Comment on the difference in graphical output between `plot(t, xt)` and `plot(xt)`. Why do you suppose this is?
- Section 1.6.1: Provide the command that adjusts the viewing region depicted in Fig. 1.6.
- Section 1.6.2: Explain why the graph of $x(t)$ in Fig. 1.7 appears jagged.
- Section 1.6.2: Comment on the graphical output of running the `stem` command for a signal consisting of a large number of elements.

- Section 1.6.2: Comment on the type of plot produced by the `scatter` function.
- Section 1.8.1: Comment on the result of plotting a function after explicitly instantiating the *figure* and *axes*.
- Section 1.8.1: Comment on what you see after plotting within `figure(2)`.
- Section 1.8.1: Modify `multiple_figures.m` to add titles and x-axis labels to each figure. Include the final version of `multiple_figures.m` within your report.
- Section 1.8.2: Modify the `subplots_example.m` so that the *axes* are arranged in a column rather than in a row.
- Section 1.8.3: Comment on the result of changing the second `plot` command to `scatter` in `multiple_plots.m`.
- Section 1.8.3: Comment on what happens when `clearvars; close all; clc;` is removed from `multiple_plots.m` and a slight modification to `x` or `y`.
- Section 1.8.4: Comment on the plots of the real and imaginary components.
- Section 1.8.4: Comment on the parametric plot in terms of real and complex exponentials. Be sure to explain why you see a spiral instead of a circle.
- Post-Lab Questions: Answer each question given.

1.R References

[1] "Read–eval–print loop," Wikipedia. (2023), [Online]. Available: https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop.

[2] "MATLAB – ans," MathWorks Inc. (2023), [Online]. Available: https://www.mathworks.com/help/matlab/ref/ans.html.

[3] "MATLAB arithmetic," MathWorks Inc. (2023), [Online]. Available: https://www.mathworks.com/help/matlab/arithmetic.html.

[4] "MATLAB constants and test matrices," MathWorks Inc. (2023), [Online]. Available: https://www.mathworks.com/help/matlab/constants-and-test-matrices.html.

[5] "MATLAB elementary math," MathWorks Inc. (2023), [Online]. Available: https://www.mathworks.com/help/matlab/elementary-math.html.

[6] "MATLAB documentation," MathWorks Inc. (2023), [Online]. Available: https://www.mathworks.com/help/.

[7] "Binomial coefficient," Wikipedia. (2023), [Online]. Available: https://en.wikipedia.org/wiki/Binomial_coefficient.

[8] "Syntactic sugar," Wikipedia. (2023), [Online]. Available: https://en.wikipedia.org/wiki/Syntactic_sugar.

[9] "MATLAB – colon," MathWorks Inc. (2023), [Online]. Available: https://www.mathworks.com/help/matlab/ref/colon.html.

[10] "MATLAB – rdivide," MathWorks Inc. (2023), [Online]. Available: https://www.mathworks.com/help/matlab/ref/rdivide.html.

[11] "MATLAB – power," MathWorks Inc. (2023), [Online]. Available: https://www.mathworks.com/help/matlab/ref/power.html.

[12] "Map (higher-order function)," Wikipedia. (2023), [Online]. Available: https://en.wikipedia.org/wiki/Map_(higher-order_function).

[13] "Functional programming," Wikipedia. (2023), [Online]. Available: https://en.wikipedia.org/wiki/Functional_programming.

[14] "MATLAB character array," MathWorks Inc. (2023), [Online]. Available: https://www.mathworks.com/help/matlab/ref/char.html.

[15] "MATLAB types of MATLAB plots," MathWorks Inc. (2023), [Online]. Available: https://www.mathworks.com/help/matlab/creating_plots/types-of-matlab-plots.html.

Sampled Sounds, Echo and Reverberation

Introduction to Audio Processing in MATLAB and Simulink

Overview

In this lab exercise, students will use MATLAB to load and plot the sampled audio data, `frere_jacques_8k.wav` and `three_blind_mice_8k.wav` (provided in this exercise's *supplementary materials*). Students will be presented with an explanation of the auditory phenomena of *echo* and *reverberation* and how linear, time-invariant systems can model such phenomena. Students will be introduced to the Simulink software and use it to modify the `echo_system.slx` and `reverb_system.slx` Simulink projects (provided in this exercise's *supplementary materials*). In addition to a typed report, students are asked to submit a completed MATLAB script, `sound_example.m`, an audio file, `my_round.wav`, and a Simulink system, `echo2_system.slx`.

Learning Objectives

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

1. Index vectors and matrices in MATLAB.
2. Slice vectors and matrices in MATLAB.
3. Load sampled audio files in MATLAB.
4. Mathematically relate a signal's sampling period to its sampling frequency.
5. Use sampling frequency to convert between a given number of samples and the corresponding time elapsed.
6. Use MATLAB to generate a time vector for a corresponding sampled signal.
7. Model the sonic phenomena of echo and reverb using discrete-time LTI system equations.
8. Use Simulink to implement discrete-time gain/delay LTI systems such as echo and reverb.

2.1 Introduction

This lab covers two main topics: working with sampled sounds in MATLAB and simulating echo and reverb systems in Simulink. First, we will explore how sounds are represented and played back within MATLAB. To this effort, we will first provide some MATLAB background on the topic of vector/matrix *indexing* and *slicing* followed by an overview of *sampling* and the ever-important concepts of *sampling periods* and *sampling frequencies*. Next, we will explore how the sonic phenomena of *echo* and *reverb* can be modeled as simple linear time-invariant systems within the Simulink software.

2.2 MATLAB Background

2.2.1 Indexing and Slicing Vectors/Matrices

We discussed constructing and generating vectors/matrices in the previous lab exercise. In this lab exercise, we will introduce the concepts of *indexing* and *slicing* vectors/matrices. The concept of indexing and array[1] may already be familiar to you from other programming languages. However, MATLAB indexes arrays (vectors/matrices) quite differently, so it's worth covering all the general cases.

Indexing Vectors

Indexing a vector or matrix allows you to retrieve elements contained within said vector/matrix. These elements are said to be located at a designated *index*. Consider the row vector:

$$\mathbf{x} = \begin{bmatrix} 10, & 9, & 8, & 7, & 6, & 5 \end{bmatrix}. \tag{2.1}$$

Create a new MATLAB session (e.g., clear all the variables, etc.) and navigate to the **Command Window**. Enter the following command to initialize $\mathbf{x}$ from Eq. (2.1):

```
x = 10:-1:5;
```

Suppose we wish to retrieve the element, 10, from the row vector `x`. This is the *first* element of `x`. In MATLAB, indexing a variable is done with *parenthesis* `( ... )` not brackets and the first element within a vector/matrix is given the index: 1. These aspects make MATLAB indexing syntax quite different from languages like C and JAVA. In particular, the practice of indices beginning at 1 (as opposed to 0) is the source of much frustration.[2] Enter the following command to retrieve the first element of `x`:

```
x(1)
```

MATLAB produces the following output:

```
ans =

    10
```

To retrieve the value 7 from `x`, we note that 7 is the *fourth* element of `x`. Enter the command:

```
x(4)
```

MATLAB produces the output:

```
ans =

    7
```

Imagine a scenario where you wish to retrieve a vector's *final* element. In the case of `x`, we *could* note that the length of the vector is 6 and then use the syntax `x(6)`. But what about the more general problem of retrieving the *final* element of a vector—no matter the length? One solution would be to compute the length of the vector using the `numel` command, as in the following example. Enter the command:

```
x(numel(x))
```

MATLAB produces the following output:

```
ans =

    5
```

Typing `x(numel(x))` is rather tedious, so MATLAB provides the built-in indexing keyword: `end`. Enter the command:

```
x(end)
```

MATLAB produces the following output:

```
ans =

     5
```

Suppose we wish to retrieve a vector's *second-to-last* element. We could accomplish this with the following command:

```
x(end-1)
```

MATLAB produces the following output:

```
ans =

     6
```

2.2.2 Slicing Vectors

We will now cover an *extremely powerful* extension to indexing offered by MATLAB: *slicing*. Suppose we wanted to create a new vector, `y`, which consisted of the *first four* elements of `x`. The naïve approach would be the following command:

```
y = [x(1), x(2), x(3), x(4)]
```

Indeed, MATLAB produces the following output:

```
y =

    10     9     8     7
```

However, MATLAB offers the following *slicing syntax*:

$$\mathtt{x}(j : k)$$

where j is the index of the *initial* element of the slice and k is the index of the *final* element of the slice. For example, to create a vector, `y`, consisting of the *first four* elements of `x`, enter the command:

```
y = x(1:4)
```

MATLAB produces the following output:

```
y =

    10     9     8     7
```

To create a vector, `y`, consisting of the *second* through *fourth* elements of `x`, enter the command:

```
y = x(2:4)
```

MATLAB produces the following output:

```
y =

     9     8     7
```

To create a vector, `y`, consisting of the elements of `x` *excluding* the *initial* and *final* elements, enter the command:

```
y = x(2:end-1)
```

MATLAB produces the following output:

```
y =

     9     8     7     6
```

Indexing Matrices

So far, we've discussed *indexing* and *slicing vectors* in MATLAB. *Matrices*, are handled similarly, but with some important nuances.

Consider the matrix:

$$\mathbf{A} = \begin{bmatrix} 1, & 2, & 3, & 4 \\ 5, & 6, & 7, & 8 \\ 9, & 10, & 11, & 12 \end{bmatrix}. \tag{2.2}$$

To create this matrix in MATLAB, enter the command:

```
A = [1:4; 5:8; 9:12];
```

Indexing matrices in MATLAB is done using the following syntax:

$$\texttt{A}(j,\, k)$$

where j is the *row* index and k is the *column* index. For instance, to retrieve the element, 1, from **A**, enter the command:

```
A(1,1)
```

MATLAB produces the following output:

```
ans =

    1
```

To retrieve the element, 2, from **A** (located in the *first* row, *second* column), enter the command:

```
A(1,2)
```

MATLAB produces the following output:

```
ans =

    2
```

To retrieve the element, 5, from **A**, enter the command:

```
A(2,1)
```

MATLAB produces the following output:

```
ans =
```

```
 
    5
```

In your lab report, give the command to retrieve the element, 11, from **A**.

Slicing Matrices

Slicing matrices functions similarly to slicing vectors with multiple dimensions. Enter the command:

```
A(1:2, 1:2)
```

MATLAB produces the following output:

```
ans =

    1    2
    5    6
```

Enter the command:

```
A(1:2, 1:4)
```

MATLAB produces the following output:

```
ans =

    1    2    3    4
    5    6    7    8
```

Like vectors, the special `end` keyword refers to the *final* index of the current dimension. When `end` is used in the *row specification*, it refers to the *total number of rows*. For example, enter the command:

```
A(1:end, 1:2)
```

MATLAB produces the following output:

```
ans =

    1    2
```

```
    5   6
    9  10
```

When used in the *column specification*, it refers to the *total number of columns*. For example, enter the command

```
A(1:2, 1:end)
```

MATLAB produces the following output:

```
ans =

    1   2   3   4
    5   6   7   8
```

A common task in MATLAB is extracting single rows or columns from a matrix. To retrieve the *first row* of **A**, enter the command:

```
A(1, 1:end)
```

MATLAB produces the following output:

```
ans =

    1   2   3   4
```

Typing `1:end` for such a common operation becomes tedious. In this case, MATLAB provides the handy short-form notation:

a *single colon* (`:`) is equivalent to `1:end`.

Returning to the example of extracting the *first row* from **A**. Enter the command:

```
A(1, :)
```

```
ans =

    1   2   3   4
```

A common mistake for students who have experience with other programming languages is

that they will attempt to extract the *first row* from **A** by entering the command: `A(1)`. This **does not work** in MATLAB. Instead, MATLAB will use *linear indexing*[3] and select a single element: 1, from the linear index, 1. Don't worry about how linear indexing works for now.

In your lab report, provide the MATLAB commands to

- retrieve the *second row* of **A**, and
- retrieve the *first column* of **A**.

2.3 Processing Digital Audio Files

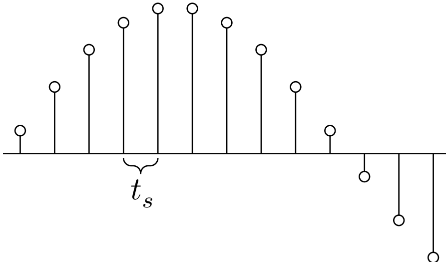


Figure 2.1: Illustration of the *sampling period*, t_s, as it relates to a discrete signal.

Discretizing an analog signal is called *sampling*. Typically, sampling occurs at a regular interval called the *sampling period*, denoted, t_s, whose unit is seconds. A simple way to think about this is that the time between any two *consecutive* samples is t_s. This relationship is illustrated in Fig. 2.1. In this figure, there are 13 samples depicted, and the time between samples is t_s. Therefore, the length of the signal snippet depicted (in seconds) is $13 \cdot t_s$.

Sometimes, rather than being given a *sampling period*, t_s, you are instead given a *sampling frequency*, denoted f_s, whose unit is hertz (Hz). The *sampling period*, t_s, is simply defined as the reciprocal of the *sampling frequency*, f_s:

$$t_s = 1/f_s.$$

In this way, if you are given t_s, you get f_s 'for free' and vice versa.

On computers, audio signals are represented as discrete-time signals. For the computer to

correctly playback these audio signals, it must know the *sampling period/frequency*. Consequentially, audio files store both the signal data and that signal's *sampling frequency*. This ensures proper playback.

2.3.1 Working with Audio in MATLAB

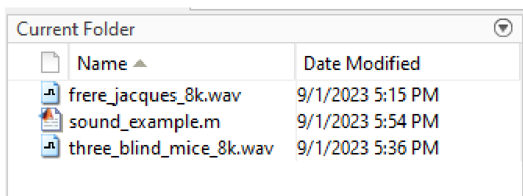


Figure 2.2: Current Folder View with both `.wav` files visible.

There are two audio files in this lab exercise's *supplementary materials*: `frere_jacques_8k.wav` and `three_blind_mice_8k.wav`. Create a new directory on your PC to serve as this exercise's project folder. Place `frere_jacques_8k.wav` and `three_blind_mice_8k.wav` in this directory. Create a new MATLAB script called `sound_example.m` and place it in this same directory. In MATLAB, check the **Current Folder View** and ensure that you can see the audio files `frere_jacques_8k.wav` and `three_blind_mice_8k.wav` (as illustrated in Fig. 2.2).

In the contents of `sound_example.m` add:

Listing 2.1: `sound_example.m`

```
1 clearvars; close all; clc;
2
3 [X, Fs] = audioread('frere_jacques_8k.wav', 'double');
4 Ts = 1/Fs;
```

Notice the command `audioread`. This command takes two parameters: the first is a *character array* containing the path to an audio file to load, and the second is a *character array* encoding the *datatype* to encode the loaded audio signal. The first parameter is `'frere_jacques_8k.wav'`, this loads the file `frere_jacques_8k.wav`. Because this file is in our *current working directory* (and is consequentially visible in the **Current Folder View**, see Fig. 2.2), it is not necessary to specify a full file path. The second parameter, `'double'`, specifies that the audio signal should

be loaded as a double-precision floating-point number.[4] On the final line of Listing 2.1, we calculate the *sampling period*, t_s (called `Ts` in our script), from the *sampling frequency*, f_s (called `Fs` in our script).

Notice that on the left-hand side of the `audioread` function, we see some new syntax: `[X, Fs]`. This syntax is used for functions that produce *multiple return values*. This is because the `audioread` function doesn't produce *one return value*; it produces *two*. This is confirmed using the `help` command on the `audioread` function. The first *return value*, which we've assigned to `X`, contains a matrix consisting of the signal data for each audio *channel* stored in the file. The second *return value*, `Fs`, is a scalar containing the *sampling frequency*, f_s, of the audio signal. This is to be expected, as proper playback requires knowledge of the *sampling frequency/period*. Keep in mind that `audioread` does *not* return a *vector*—it returns two *return values*. Even though the brackets in `[X, Fs]` are suggestive of vector initialization, they are completely different syntactically.

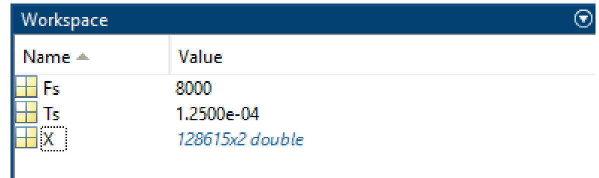


Figure 2.3: Variable Workspace after loading the audio file `frere_jacques_8k.wav`.

Run the script with your entered code and note the **Variable Workspace**. It should appear similar to Fig. 2.3. The **Variable Workspace** is a valuable tool for inspecting each of the variables you've initialized in your script. For instance, it tells us about the audio data loaded from `frere_jacques_8k.wav`. We can see that the sampling frequency, f_s, is 8 kHz. We can also see that the signal data, stored in `X`, is a *two column* matrix with many rows (in my case, the number of rows is $128,615$; this may be different for you depending on the version of this lab exercise).

We can *inspect the contents* of `X` by double-clicking on the box icon next to `X` in the **Variable Workspace**. This should pull up a **Variable Viewer** tab. This viewer depicts the contents of `X`—formatted as a spreadsheet (as illustrated in Fig. 2.4). Again, we can see that the matrix, `X`, consists of *two columns* with many rows. To understand how to interpret this matrix, use the

X

128615x2 double

	1	2	3	4	5	6	7
1	-0.00100708...	-0.00100708...					
2	0.001190185...	0.001159667...					
3	1.831054687...	2.441406250...					
4	0.003417968...	0.003479003...					
5	0.004333496...	0.004364013...					
6	0.007263183...	0.007354736...					
7	0.007293701...	0.007385253...					
8	0.010650634...	0.010711669...					

Figure 2.4: Contents of the `X` matrix. There are additional rows beyond 8 (not pictured).

`help` command on the function `audioread` in the **Command Window**. Inspecting the contents of variables like this is often a useful way to confirm that you're producing the results that you're expecting. When you're done inspecting the contents of `X` you can close the **Variable Viewer** tab to return to your script.

The `help` command explains that `audioread` loads the audio signal for each *channel* of the audio file into a matrix, `X`, where each column of `X` contains an individual channel of the audio signal and the number of rows is the number of samples within each channel. In our case, `frere_jacques_8k.wav` is a *stereo signal*, this means that the audio file contains two separate audio *channels*: one for the left ear, and another for the right ear. It is standard for the first channel within a stereo sound file to denote the left channel while the second denotes the right.

For this lab, we only want to work with a single signal from `frere_jacques_8k.wav`, so we will create a new variable, `x`, which is initialized to the left audio channel from `X`. In your `sound_example.m` script, add a new line that defines a variable, `x`, to be the first column of `X`. You'll want to include a semicolon (`;`) to suppress printing the (very long) resultant vector.

```
5 x = TODO
```

Replace `TODO` with your code to initialize `x`.

Next, we need to generate a companion *time vector* for `x`, called `t`. This time vector should:

- begin at time 0 (seconds),
- contain the *same* number of elements as `x`, and
- the spacing between consecutive elements (the *step/increment*) should be the sampling

period of the audio file, `Ts`.

It may be useful to review Sec. 2.3. Here are some hints to help you.

- Do *not* use the `linspace` function; it's easier to use the colon operator (`:`).
- You can find the number of elements in `x` using `numel`.
- A vector generated by the command `0:6` does *not* contain 6 total elements.

Once you have figured out the proper command, add the definition of `t` to your `sound_example.m` script (don't forget the semicolon).

```
6 t = TODO
```

Replace `TODO` with your code to initialize `t`. Run the script and confirm that the total number of elements in `x` is the same as the total number of elements in `t`. You can do this using the `numel` function or by looking at the **Variable Workspace**.

Now, add code to your `sound_example.m` script to plot `x` with respect to `t`.

```
7  % plot left channel
8  figure(1)
9  plot(t, x)
10 axis([t(1), t(end), -0.15, 0.1])
11 xlabel('t (seconds)')
12 title('Frere Jacques')
```

Run the MATLAB script.

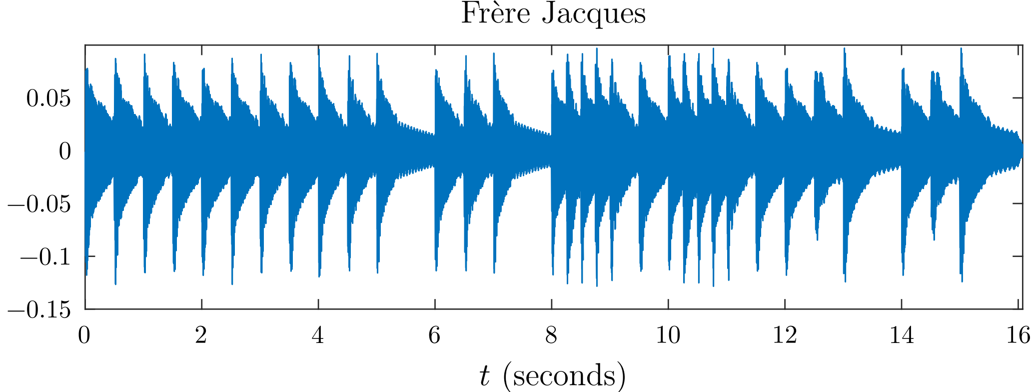


Figure 2.5: Plot of the audio signal for the left channel of `frere_jacques_8k.wav`.

If all goes well, your graph should appear similar to the one depicted in Fig. 2.5. Check that

you've calculated `t` correctly by comparing the t-axis in Fig. 2.5 to the t-axis of your graph.

Finally, we will play the audio signal represented by `x` through the computer's speakers. This is done using the `sound` function. Remember, for a computer to properly play back a discretized audio signal, you must provide it with the *sampling period* or *sampling frequency*. In this case, the `sound` function takes the *sampling frequency* as a parameter. Add the following code to your `sound_example.m` script:

```
13 % play sound through the computer speaker
14 sound(x, Fs);
```

Run the script; you should hear the melody to "Frère Jacques" playing through your computer's speaker. You may need to change your PC's audio output device if you do not; This can be done using the *volume mixer* from the Windows taskbar.

Include your completed `sound_example.m` in your submission as an additional attachment along with your typed lab report.

2.4 Modeling Echo and Reverberation

Echo and reverberation are two types of sonic phenomena that alter how sound is perceived. These concepts are nicely modeled using *linear time-invariant systems*. We will briefly introduce Simulink, a software suite capable of modeling and simulating both phenomena. Reverberation and echo occur because the listener hears two or more versions of the same sound, where each version arrives at a slightly different instant.

2.4.1 Echo

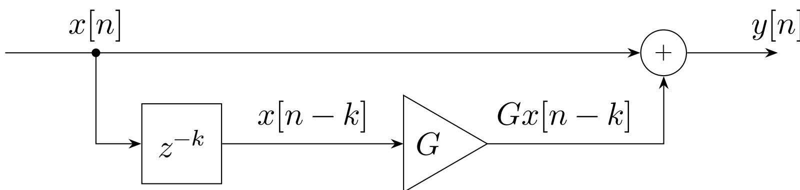


Figure 2.6: Block diagram of a basic echo system.

Echo is a sonic phenomenon in which a listener perceives multiple, distinct repetitions (in decreasing volume) of a single sonic event. It occurs when sound rebounds off nearby topological features and is generally perceivable because the speed of sound is relatively slow (about 400 meters per second). Systems producing echoes can be elegantly modeled by linear, time-invariant systems.

Keeping with our theme of digitally sampled signals, let $x[n]$ represent a digitally sampled audio signal that will serve as the input to an echo-creating system and let $y[n]$ be the output of this system. Consider a discrete-time system that takes the input $x[n]$ and introduces a single sonic repetition k-samples later with a gain of G. Such a system could be modeled by the following equation:

$$y[n] = x[n] + Gx[n-k]. \tag{2.3}$$

Here, k is an integer specifying the number of samples to delay $x[n]$ before the echo/repetition is sounded, and G is a real number specifying the amount of gain/attenuation to apply to the input $x[n]$. To achieve a realistic echo effect in which the delayed sound is reduced in volume, the gain, G should satisfy $|G| < 1$. This equation is represented graphically in Fig. 2.6. Here, the z^{-k} block denotes the "delay by k samples" operation. This notation comes from the Z-Transform,[5] which is covered in more detail later in the curriculum.

We constructed the echo equation, Eq. (2.3) as a discrete-time system with an integer delay of k-samples. However, it is desirable to describe the delay introduced by an echo system in *seconds*. In your lab report, describe what piece of information is needed to convert a delay of k-samples to τ-seconds and give the equation for doing so (i.e., $\tau = ...$). *Hint*: Think back to Sec. 2.3. What is the relationship between samples and seconds?

The echo system of Eq. (2.3) models a single attenuated delay. However, many echo systems encountered in the real world contain multiple delays. Additional echoes/delays can be added by introducing additional gain *paths*, as illustrated in Fig. 2.7. The key point is that the echo is derived solely from the input. This type of system is known as a *finite impulse response* (FIR) system.

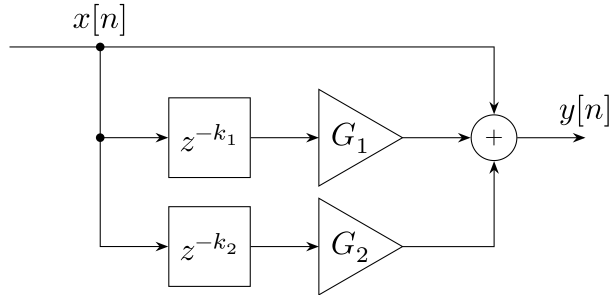


Figure 2.7: Block diagram of an echo system with multiple echo paths.

2.4.2 Reverberation

Reverberation is a sonic phenomenon similar to echo. In reverberation, a listener perceives a sonic event that has had its timbre/quality altered by reflecting off various surfaces. These surfaces will absorb certain frequencies and have their phases altered. Suppose $x[n]$ represents a digitally sampled audio signal that will serve as the input to a reverb system, then the output of that reverb system, y is given by:

$$y[n] = x[n] + Gy[n-k]. \tag{2.4}$$

Where k is an integer specifying how many samples back to take the previous output, $y[n-k]$, before it is feedback into the current output, $y[n]$, and G is a real number specifying the amount of gain/attenuation to apply to the input $x[n]$. In reverberation, the output is derived from *both* the input and the previous output: This equation is represented graphically in Fig. 2.8. In your lab report, describe the difference between echo and reverberation.

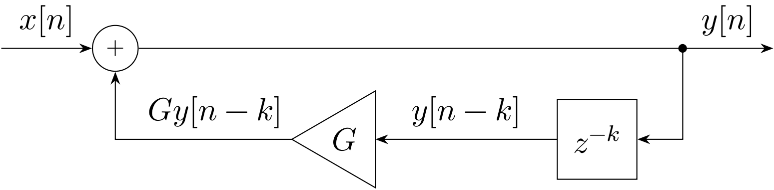


Figure 2.8: Block diagram of a reverb system.

2.4.3 Simulink

Simulink is software that comes bundled with MATLAB. According to MathWorks, "*Simulink is an environment for multidomain simulation and Model-Based Design for dynamic and embedded systems. It provides an interactive graphical environment and a customizable set of block libraries that let you design, simulate, implement, and test various time-varying systems, including communications, controls, signal processing, video processing, and image processing.*"[6]

This means Simulink can be used to quickly assemble prototypes and models of systems using a graphical user interface. Systems can be built by connecting different blocks. These blocks can be selected from pre-defined "block sets" for a particular application (e.g., communications, image processing) or can be defined by the user. User-defined blocks can be created by specifying the mathematical operations performed by the block inputs.

We will not use Simulink extensively in this course, so it will not be covered as in-depth as MATLAB. There are two Simulink files in this lab exercise's *supplementary materials*: `echo_system.slx` and `reverb_system.slx`. Copy these files into your project folder for this lab exercise. The two audio files `frere_jacques_8k.wav` and `three_blind_mice_8k.wav` should be in this same folder. Launch Simulink by typing `simulink` in the MATLAB **Command Window**.

2.4.4 The Echo System

Once Simulink is loaded, launch the file `echo_system.slx` by double-clicking it.

You should see a window featuring a block diagram to the one depicted in Fig. 2.9. This block diagram describes the *echo* system given by Eq. (2.3). Compare this to the block diagram given in Fig. 2.6. The input to the system is `frere_jacques_8k.wav`. This is represented by the **Input Block** with the text `frere_jacques_8k.wav`, and `A: 8000 Hz, 16bit, stereo`.

If you do not see the second piece of text, the `.wav` file did not load properly. In this case, double-click the **Input Block**. A new window will pop up, and there will be an option to browse for a sound file. Click *browse* and navigate to and select `frere_jacques_8k.wav`.

The system's output is your computer's speaker, represented by the speaker-shaped **Output Block**. The gain, G from Eq. (2.3), is represented by the triangular **Gain block**; currently, the gain is set to 0.7. Finally, the delay, k from (2.3), is represented by the rectangular **Delay**

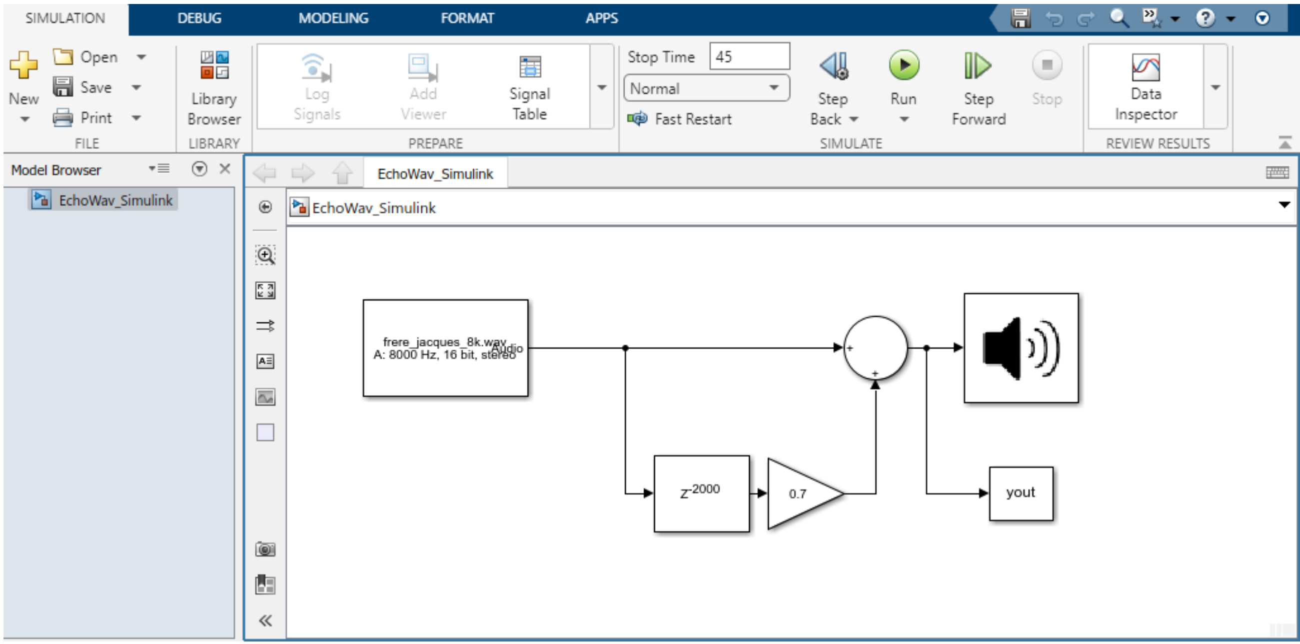


Figure 2.9: Simulink diagram for an echo system.

Block with the text z^{-2000}—indicating that the delay is set to $k = 2000$ samples. As mentioned before, the notation, z^{-k}, comes from the Z-Transform.[5]

Run the model from the upper menu by clicking the **Run Simulation** button (green arrow). Verify that you can hear the echo. This model has one input, the sound file `frere_jacques_8k.wav`, and two outputs—one to an audio port and one a variable called `yout` in the MATLAB workspace. In Simulink, you can click on any block to view and set its parameters.

Modifying the Delay-Time and Gain

Double-click on the **Delay Block** and change the *Delay length* to a different value (see Fig. 2.10). This delay is specified in *samples*, not in *seconds*. Recall from Sec. 2.3 that you can convert between the *number of samples* and *seconds* if you have knowledge of either the sampling period, t_s, or the sampling frequency, f_s. Fortunately, the input block tells you the sampling frequency, f_s. Run the model and hear how the effect has changed. Try a few different settings between 100 and $20,000$ samples.

Double-click the **Gain Block** and change the *Gain* to a different value. Run the model and hear how the effect has changed. Try a few different settings between 0 and 2. Note how the

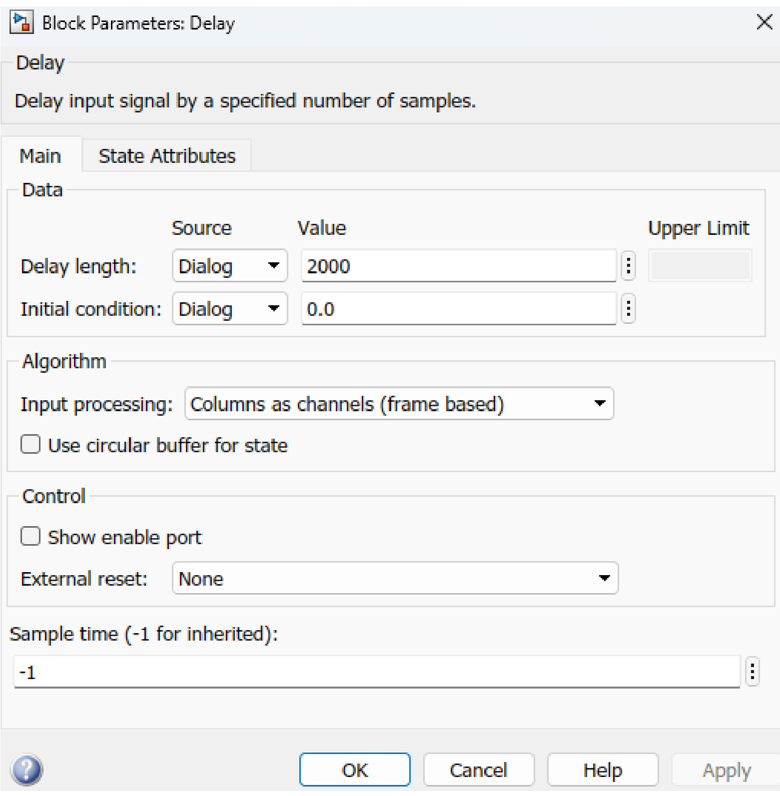


Figure 2.10: Parameter configuration dialog of a **Gain Block** in Simulink.

output changes. Make a table of the gain and delay variables you tried and include it in your lab report. This table should include at least 5 total entries with at least 3 unique delay values and at least 3 unique gain values.

Investigate the **Delay Block** more deeply. In your lab report, estimate the smallest delay value that allows you to perceive an echo. Investigate the **Gain Block** more deeply. In your lab report, describe what happens when the gain value approaches 1 and what happens when the gain value exceeds 1. Explain what causes the phenomenon you observe.

2.4.5 The Reverb System

Select the **Open** button from the **Simulation** tab from the upper menu in Simulink. This will launch the **Open Project Dialog**. Locate and open `reverb_system.slx` from this dialog.

You should see a window similar to the one depicted in Fig. 2.11. This block diagram describes the *reverb* system given by Eq. (2.4). Compare this to the block diagram given in Fig. 2.8. As before, ensure that the second row of text, `A: 8000 Hz, 16bit, stereo`, is visible on the **Input**

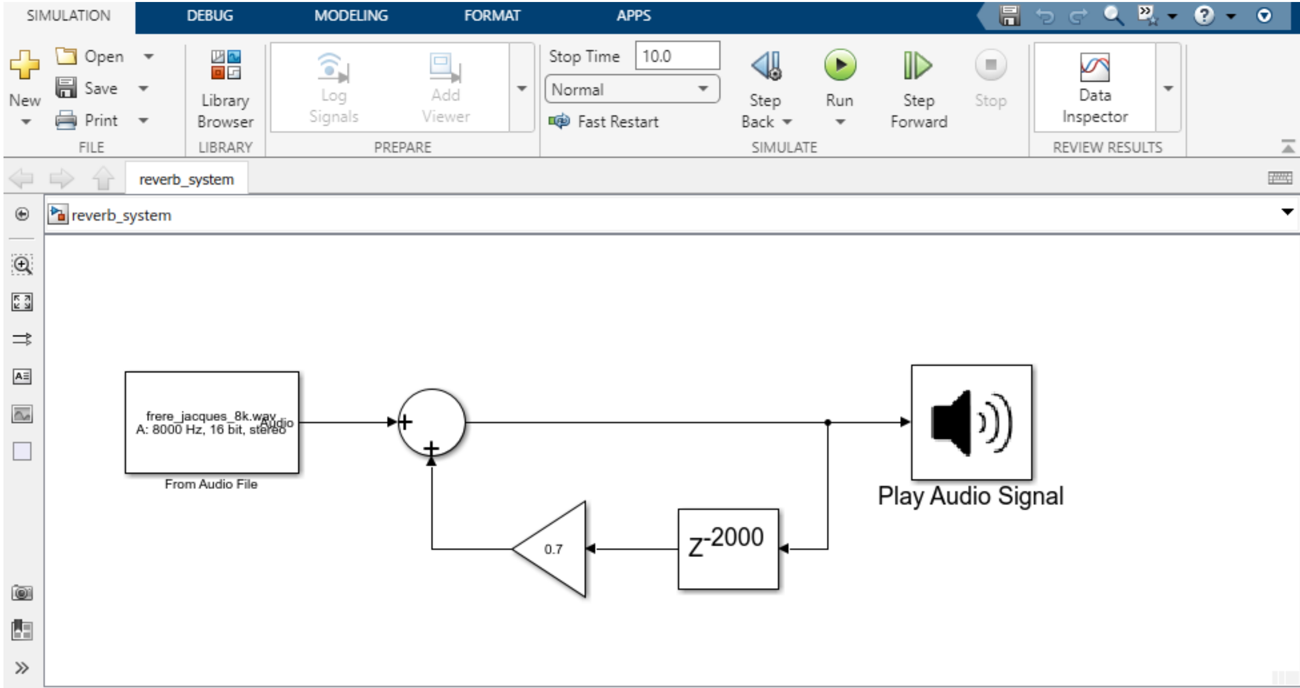


Figure 2.11: Simulink diagram of a reverb system.

Block; this ensures that the `.wav` was loaded correctly. Run the model and verify that you can perceive the reverb effect.

Modifying the Delay-Time and Gain

Increase the delay on the **Delay Block**. Run the model and take note of the effect. Try a few different delay values. Does reverberation require a shorter delay time than echo to be noticeable? Answer this in your lab report. Provide some reasoning as to why this may (or may not) be.

Double-click the **Gain Block** and change the *Gain* to a different value. Run the model and hear how the effect has changed. Try a few different settings between 0 and 2. Note how the output changes. In your lab report, describe the effect of the gain value on the system's stability. Give a value for which the system is *stable* and a value for which the system is *unstable*.

2.4.6 Singing Rounds

Reload the original version of `reverb_system.slx`. Set the input file to `frere_jacques_8k.wav` or `three_blind_ mice_8k.wav`. You can do this by double-clicking the **Input Block** and clicking the *browse* button from the prompt that opens—from here, you can navigate to the `.wav` file that you wish to select. Now we will adjust the delay such that the simulation plays as a *round*.[7]

A *round* is a type of melody in which multiple people sing in an overlapping manner. The first person begins singing the melody alone. After a delay, another person will join in—singing on top of the first. Then, after another delay, another person will join in—and so on. A popular round that many are familiar with is "Row, Row, Row Your Boat." The `.wav` files provided ("Frère Jacques" and "Three Blind Mice") are traditionally sung as *rounds*. Rounds are typically constructed with a particular time delay in mind. Selecting this time-delay will make each instance of the melody form a harmony between one another. Usually, this delay amount corresponds to the time it takes to sing one measure of the melody.

Within the `reverb_system.slx`, see if you can adjust the delay to maximize how "harmonious" the round sounds in simulation. While the best delay value is somewhat subjective, the amount should be *at least* a few seconds long. Specify the delay amount (in seconds) in your lab.

You can save your output file by adding a **To Multimedia File** block in the DSP signal processing toolbox under the subdirectory **Sinks**. This can be accessed through the **Upper Menu**. Ensure that the **Simulation** tab is selected and click the **Library Browser** button. In the dialog that pops up, navigate to `DSP System Toolbox-> Sinks` (see Fig. 2.12). Once you've found the optimal time delay to create a round, save your audio output as `my_round.wav`. In `my_round.wav` as an additional attachment in your lab submission.

2.4.7 Creating a Realistic Echo Subsystem

Reload `echo_system.slx`. Now modify this system to create as realistic an echo as possible (multiple repetitions of decreasing volume). Create additional echo paths using the Simulink block library in the DSP System Toolbox. Open the **Library Browser** from the **Simulation** tab of the **Upper Menu**. Select the menu item `Library Browser -> Commonly Used Blocks` to find the **Gain Block** and **Delay Blocks**. Once you have achieved a result you are satisfied with,

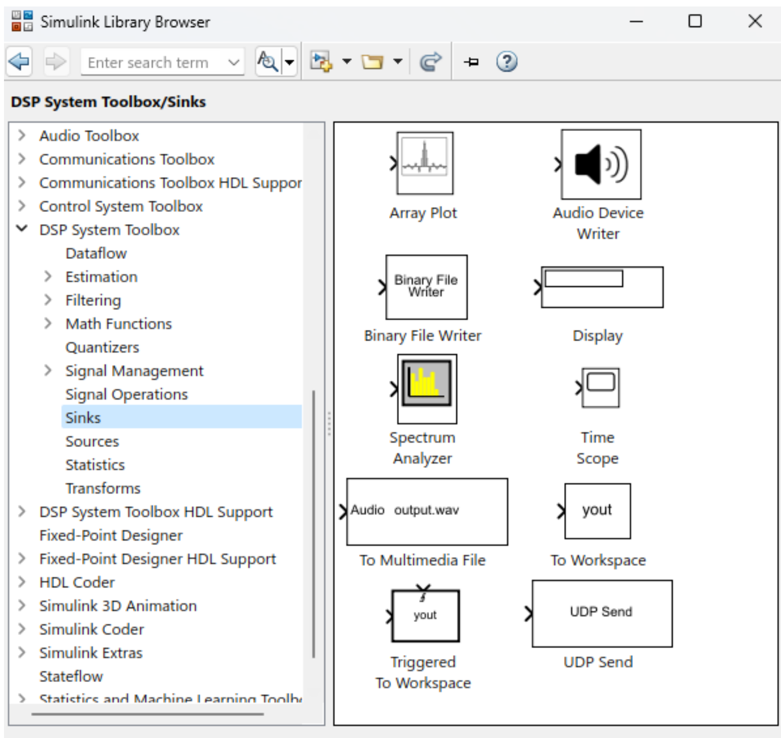


Figure 2.12: The Block Library Browser.

save your Simulink system as `echo2_system.slx`. Include `echo2_system.slx` as an attachment to your submission. In your lab report, discuss the rationale behind the *gain* and *delay* you chose.

2.5 Post-Lab Questions

Q.1

A student types the following command: `y = x(0:3);`

Why is this command incorrect?

Q.2

Suppose `X` is a matrix. How does one assign:

- the vector, `p`, as the *first column* of `X`,
- the vector, `q`, as the *second column* of `X`, and
- the vector, `s`, as the *first row* of `X`?

Q.3

Mathematically speaking, what is the main difference between echo and reverberation?

Q.4

Why does the slow speed of sound mean that echo can be perceived? What would happen if the speed of sound was much faster?

Q.5

Consider a discrete-time echo system operating at a sampling rate of 8 kHz. Suppose this system satisfies the following equation: $y[n] = x[n] + 0.5x[n - 1500]$. What is the time duration (in seconds) of the delay introduced by the $x[n - 1500]$ term?

2.6 Report Checklist

Work with lab partner(s) to create a lab report for this lab exercise. Be sure to include the names of each person in your group on the report. This report should be a typed document that outlines the basic procedure of the lab exercise and should include any results you were asked to obtain. Your report should follow a *cohesive narrative*—it should always be apparent *why* you have presented the results you have. Do not simply insert results without (1) motivating the problem and (2) providing an analysis of the data. Ensure that your report is grammatically correct with complete sentences. This does not mean your report must be wordy with filler—in fact, avoid filler and state your ideas concisely.

The final Post-Lab Questions section does need to be woven into your lab narrative. You may simply append your answers to each question at the end of your report in an attached answers section. Be sure to indicate which question you are answering by referencing the corresponding question number.

Be sure the following are included in your report:

- Section 2.2.2: Give the command to retrieve the element, 11, from **A** (given by Eq. (2.2))
- Section 2.2.2: Provide the MATLAB commands to retrieve the *second row* and *first column* of **A** from Eq. (2.2).
- Section 2.3.1: Include the completed `sound_example.m` as an additional attachment in your

submission alongside your typed report.

- Section 2.4.1: Describe what information is needed to convert a delay of k-samples into τ-seconds and give the equation for doing so (i.e., $\tau = ...$).
- Section 2.4.2: Describe the difference between echo and reverberation.
- Section 2.4.4: Create a table of the gain and delay variables. This table should have at least 5 total entries with at least 3 unique delay values and at least 3 unique gain values.
- Section 2.4.4: Estimate the smallest delay value that allows you to perceive an echo.
- Section 2.4.4: Describe the phenomenon that occurs when the gain approaches, then exceeds 1. Explain what causes this phenomenon.
- Section 2.4.5: Determine whether reverb requires shorter delay times than echo to be noticeable. Provide some reasoning as to why this may (or may) be the case.
- Section 2.4.5: Describe the effect of the gain value on a reverb system's stability. Give a value for which the stable is *stable* and a value for which the system is *unstable*.
- Section 2.4.6: Specify the delay value (in seconds) you chose to create a round. Specify which `.wav` file you chose as your input.
- Section 2.4.6: Include the audio file `my_round.wav` containing your implemention of a *round* featuring either "Frère Jacques" or "Three Blind Mice" as an additional attactment in your submission.
- Section 2.4.7: Include your realistic echo subsystem, `echo2_system.slx`, as an attachment to your submission.
- Section 2.4.7: Discuss the rationale behind the *gain* and *delay* you chose for your realistic echo subsystem.
- Post-Lab Questions: Answer each question given.

2.7 Acknowledgments

This lab's echo/reverb/Simulink section is based on a lab brief originally written by **Stewart Worrall**, **Simon Henley**, and **Philip Jackson** in co-operation with **Texas Instruments**.[8]

2.R References

[1] "Array (data structure) – element identifier and addressing formulas," Wikipedia. (2023), [Online]. Available: http://en.wikipedia.org/w/index.php?title=Array_(data_structure)&oldid=1170526080#Element_identifier_and_addressing_formulas.

[2] Anoop. "Why does matlab have 1 based indexing," Stack Overflow. (Mar. 2014), [Online]. Available: http://stackoverflow.com/questions/22546787/why-does-matlab-have-1-based-indexing.

[3] "MATLAB array indexing," MathWorks Inc. (2024), [Online]. Available: https://www.mathworks.com/help/matlab/math/array-indexing.html.

[4] "Double-percision floating-point format," Wikipedia. (2023), [Online]. Available: http://en.wikipedia.org/w/index.php?title=Double-precision_floating-point_format&oldid=1168913577.

[5] "Z-transform," Wikipedia. (2023), [Online]. Available: http://en.wikipedia.org/w/index.php?title=Z-transform&oldid=1173089093.

[6] "Simulink documentation," MathWorks Inc. (2023), [Online]. Available: http://www.mathworks.com/help/simulink/.

[7] "Round (music)," Wikipedia. (2023), [Online]. Available: http://en.wikipedia.org/w/index.php?title=Round_(music)&oldid=1162426673.

[8] S. Worrall, S. Henley, and P. Jackson. "EE1.LabB: D3 Echo and reverberation," University of Surrey. (2007), [Online]. Available: http://personal.ee.surrey.ac.uk/Personal/P.Jackson/ee1.lab/D3_echo/.

Introduction to CyDAQ and DAD

An Introduction to Lab Hardware and an Exploration of its Limitations

Overview

In this lab, students will be introduced to two pieces of lab equipment, the *Cyclone Data Acquisition System* (CyDAQ) and the *Digilent Analog Discovery 2* (DAD). Students will use the WaveForms Software (freeware from Digilent) to configure the DAD to act as a signal generator and an oscilloscope. Students will use the CyDAQ Software configuration tool to configure the CyDAQ to act as a filter and use the CyDAQ to capture sampled data. In addition to a typed report, students are asked to submit a completed MATLAB script, `plot_filtered_square.m`.

Learning Objectives

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

1. Use the WaveForms Software to configure the DAD's signal generator.
2. Use the WaveForms Software to configure the DAD's oscilloscope.
3. Use the CyDAQ Software to configure the CyDAQ's built-in filter.
4. Use the CyDAQ Software to capture sampled data.
5. Load variables stored in mat files into the MATLAB workspace.
6. Measure the rise time of a signal captured on the DAD's oscilloscope.
7. Relate the concept of rise time to maximum achievable bit rates.

Materials

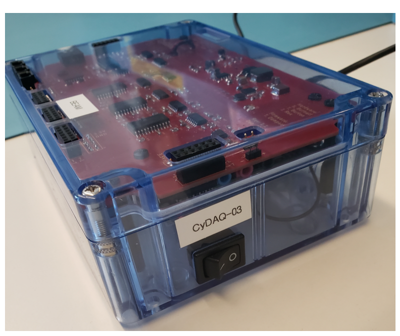
(1×) Cyclone Data Acquisition System (CyDAQ)

(1×) Digilent Analog Discovery (DAD)

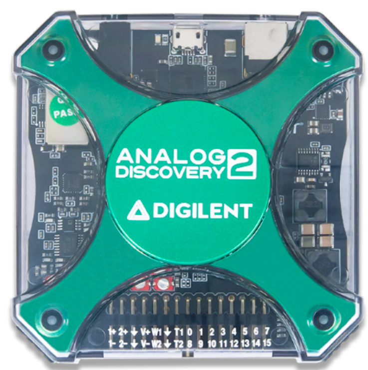
3.1 Introduction

This lab will introduce the two pieces of equipment you will use throughout the remaining lab exercises: the *Cyclone Data Acquisition System* (CyDAQ) and the *Digilent Analog Discovery 2* (DAD). The CyDAQ, depicted in Fig. 3.1a, was originally a senior design project integrated into many ECpE labs beginning. The CyDAQ is equipped with: 6 on-board filters, 5 input sources, analog output, and an in-built sampling ability. It is powered by a Zynq Z7 FPGA[1] with an onboard ARM 667 MHz dual-core Cortex-A9 processor. It is controlled and configured through the `CyDAQ Software`. In this and future lab exercises, we will use the CyDAQ to *filter signals* and *capture data*.

The DAD,[2] depicted in Fig. 3.1b, is a USB device which contains two built-in signal generators and two built-in oscilloscope channels. It is controlled and configured using the `WaveForms software`. A full breakdown of the specifications is available through Digilent's website.[3] In this and future lab exercises, we will use the DAD as a *signal generator* and *oscilloscope*.



(a) The CyDAQ with power switch visible on the right side.



(b) The Digilent Analog Discovery 2 (DAD).

Figure 3.1: The CyDAQ and DAD devices.

The standard configuration, which we will return to repeatedly in these lab exercises, is depicted in Fig. 3.2. The signal flow through this configuration is as follows:

1. a signal generated by the DAD is fed into the CyDAQ's V_{in} terminal,
2. the CyDAQ will perform a filtering operation to this signal,
3. the signal is fed out the CyDAQ's V_{out} terminal and into scope probes on the DAD.

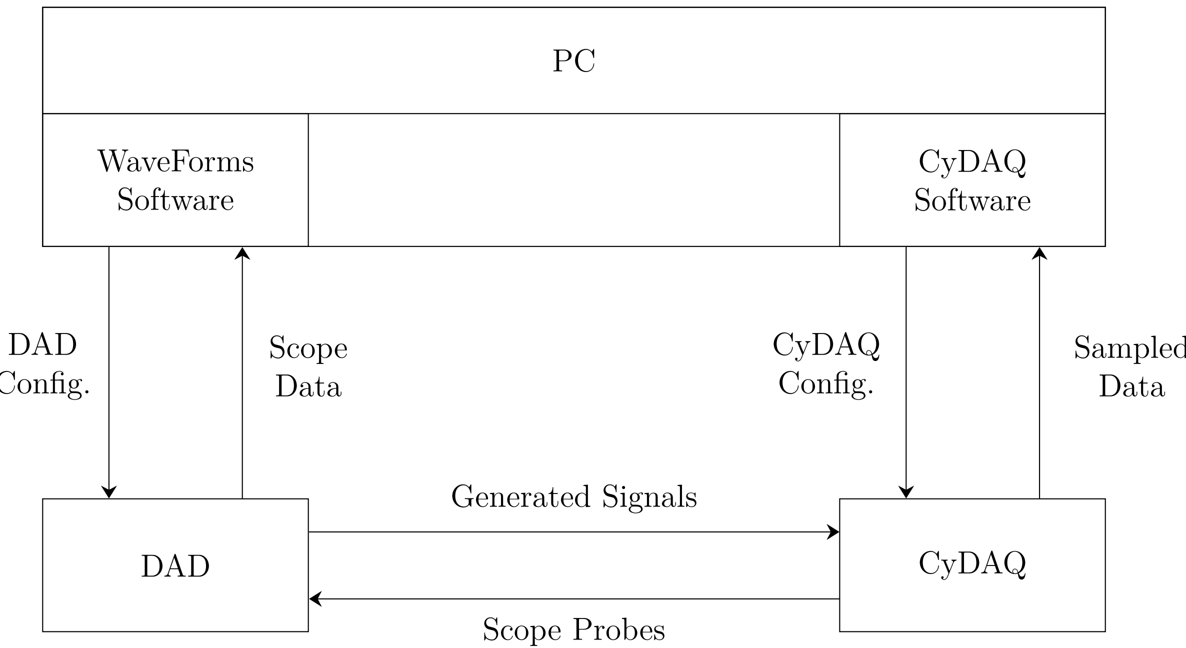


Figure 3.2: Illustration of how the CyDAQ, DAD, and PC connect and interact with one another.

This can be simplified further to: `Signal Gen (DAD)` ▶ `Filter (CyDAQ)` ▶ `Scope (DAD)`.

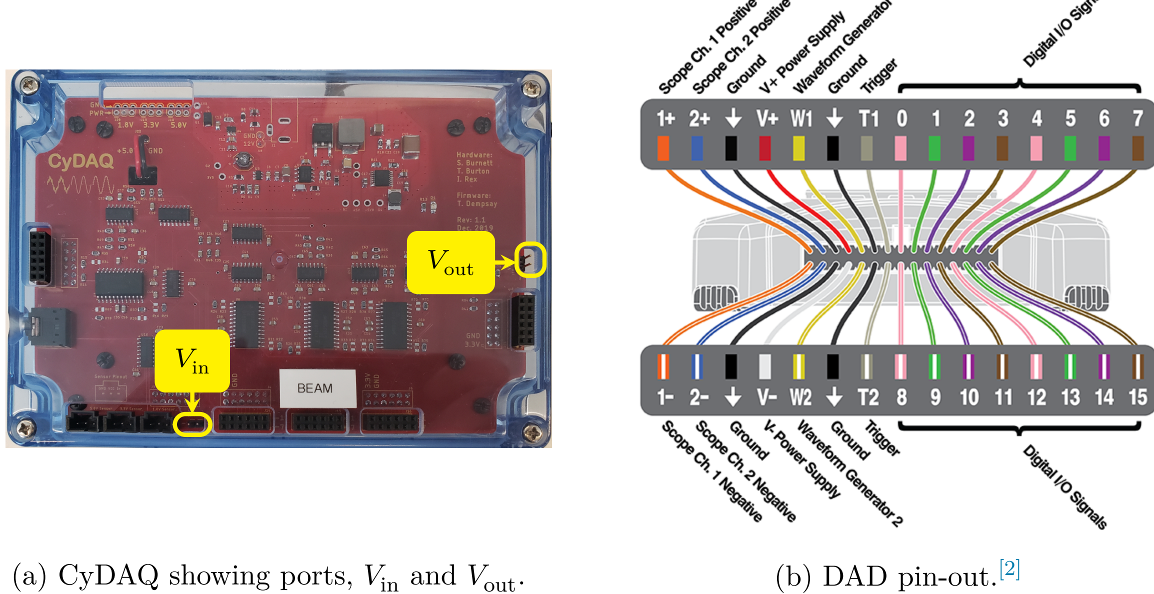
3.2 Connections and Setup

Now, we will connect the hardware to implement the configuration described in the section above. First, locate the **Waveform Generator 1** (`W1`) and **Ground** (`GND`) jumper wires on the DAD—see Fig. 3.3b. These constitute the output of the first waveform generator. Locate the two, male, V_{in} terminals on the CyDAQ (J9)—see Fig. 3.3a. You should see that the leftmost terminal is marked negative (`-`) and the rightmost terminal is marked positive (`+`). We shall call these `Vin-` and `Vin+` respectively.

- Connect the **DAD's Waveform Generator 1** (`W1`) and the **DAD's Ground** (`GND`) **Jumper Wires** to the **CyDAQ's V_{in} Terminals**, (`Vin+`) and (`Vin-`), respectively.

Next, locate **Scope Ch. 1 Negative** (`CH1-`) and **Scope Ch. 1 Positive** (`CH1+`) on the DAD. These constitute the first oscilloscope on the DAD. Locate the two male, V_{out} terminals on the CyDAQ. You should see that the lower terminal is marked negative (`-`) and the upper terminal is marked positive (`+`). We shall call these `Vout-` and `Vout+` respectively.

- Connect the **DAD's Scope Ch. 1 Terminals**, (`CH1-`) and (`CH1+`) to the **CyDAQ's V_{out} Terminals**, (`Vout-`) and (`Vout+`), respectively.

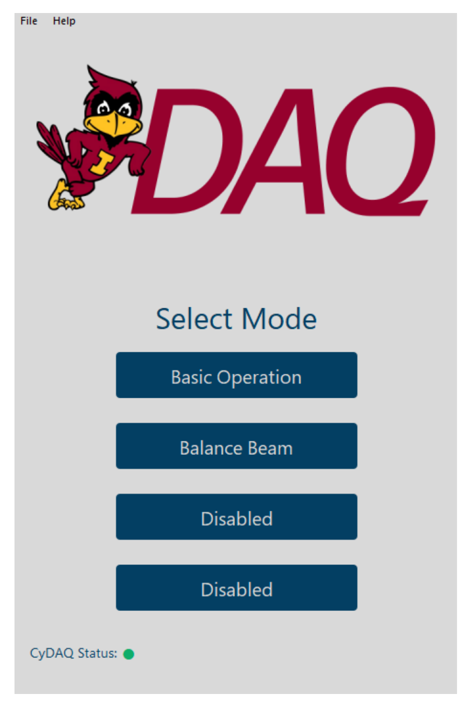


(a) CyDAQ showing ports, V_{in} and V_{out}.

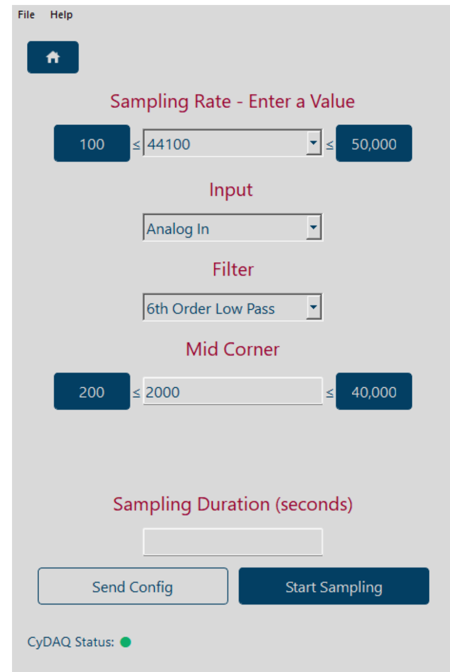
(b) DAD pin-out.[2]

Figure 3.3: Pin-outs/terminals for CyDAQ and DAD.

3.3 CyDAQ Software



(a) Main screen.



(b) Basic settings screen.

Figure 3.4: CyDAQ configuration software.

As mentioned, these lab exercises will use the CyDAQ to *filter signals* and *capture data*. To learn the basics of the CyDAQ and DAD, we will first configure the CyDAQ to act as a pass-through filter. This filter passes the input directly to the output—leaving the underlying signal unchanged. Configuration of the CyDAQ (as well as data capture) is done using the `CyDAQ Software`. This software has been installed on each lab machine and can be located by searching "CyDAQ" in the Windows start menu. Upon launching the software, you should see the **Main Screen**—depicted in Fig. 3.4a.

Once on the **Main Screen**, wait a few seconds and confirm that the CyDAQ connection indicator is green (●). If it is not green:

- ensure that your CyDAQ is turned on—the power switch is on the right side (see Fig. 3.1a),
- ensure that the 5.5 mm power cable is connected at the back of the CyDAQ, and finally
- ensure that the CyDAQ is connected to the PC via USB.

Once you have confirmed that your CyDAQ is connected, click the **Basic Operation** button

to open the **Basic Settings Screen**—depicted in Fig. 3.4b. For a pass-through configuration, set:

- **Sampling Rate**: 48000 Hz
- **Input**: Analog In
- **Filter**: All Pass.

Then click the **Send Config** button at the bottom of the **Basic Settings Screen**. This will push these settings to the CyDAQ.

3.4 WaveForms Software

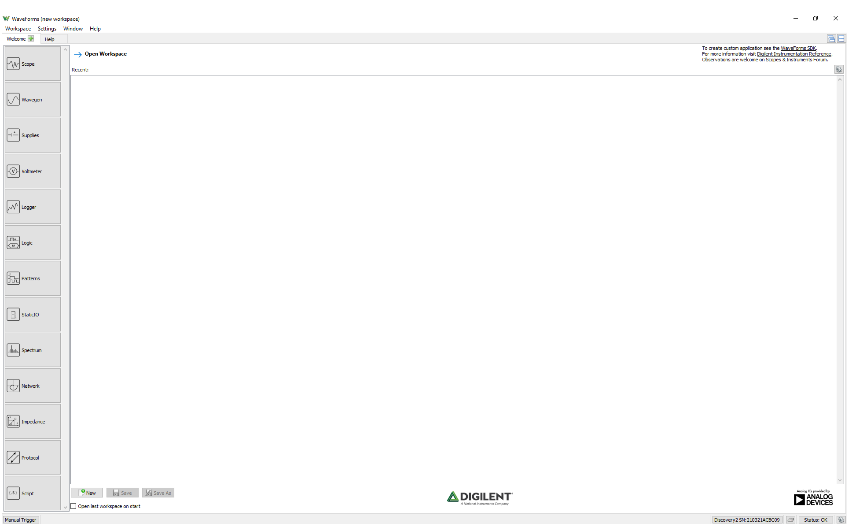


Figure 3.5: WaveForms home page

As mentioned above, these lab exercises will use the DAD as a *signal generator* and *oscilloscope*. The DAD is controlled via the `WaveForms Software`. This software is freely available from Digilent,[4] however, this software has already been installed at each lab station and can be located by searching "WaveForms" in the Windows start menu.

Upon launching WaveForms, you should see a window similar to the one depicted in Fig. 3.5. Take a brief look at the options and select **Wavegen**. This should open a window similar to the one depicted in Fig. 3.6. This is the **Wave Generator** tab. Here, we can construct signals that will be output through the DAD's output connection (`W1`). In the **Wave Generator**, configure the following signal:

- **Type**: Sine
- **Frequency**: 10 kHz
- **Amplitude**: 2 V
- **Offset**: 0 V
- **Symmetry**: 50%
- **Phase**: 0°

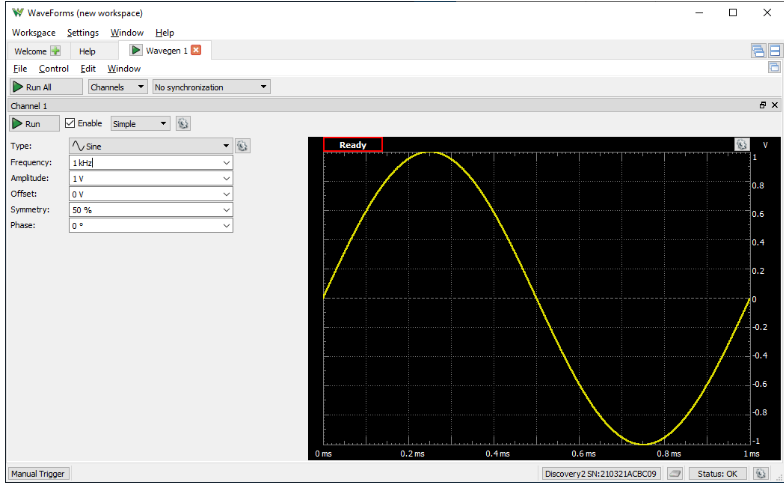


Figure 3.6: Wavegen page.

Now press the **Run All** button (next to the green play arrow) to tell the DAD to start generating the 10 kHz sinusoid we just configured. Generally, unless we are making changes to the **Wave Generator**, we will leave **Run All** on. Following our signal chain, this 10 kHz sine travels out the DAD and into the CyDAQ's input. Then, the CyDAQ outputs the signal (unaltered) to the scope probes for channel 1 on the DAD.

It is desirable to view the read-out of our channel 1 oscilloscope. To do so, we need to open a **Scope** tab. Locate the **Welcome** tab in the top-left portion of the WaveForms window. You should see a green plus sign (+). Click this plus sign and select **Scope** from the drop-down menu. This will add **Scope** tab. The contents of the **Scope** tab should be similar to Fig. 3.7. You should now have a tab for the **Wave Generator** and a tab for the **Scope** in your WaveForms instance.

In your **Scope** tab, press the **Run** button. Let the scope run for a moment, now press the **Stop** button. You should see a yellow signal as in Fig. 3.8. This yellow signal corresponds to the

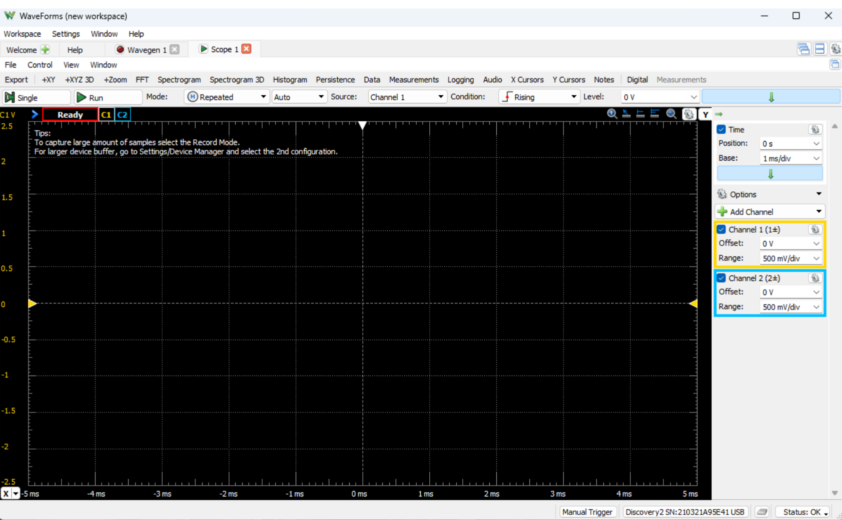


Figure 3.7: WaveForms Scope Tab.

channel 1 oscilloscope (**CH. 1**) within the DAD. In other words, the output of the CyDAQ filter (which is currently a pass-through). You may also see a blue signal. This corresponds to the channel 2 oscilloscope (**CH. 2**). You are not using this oscilloscope channel (which is probably picking up undesirable electromagnetic interference). Remove the blue signal by unchecking the blue box, labeled **Channel 2**, on the right-hand portion of the **Scope** tab.

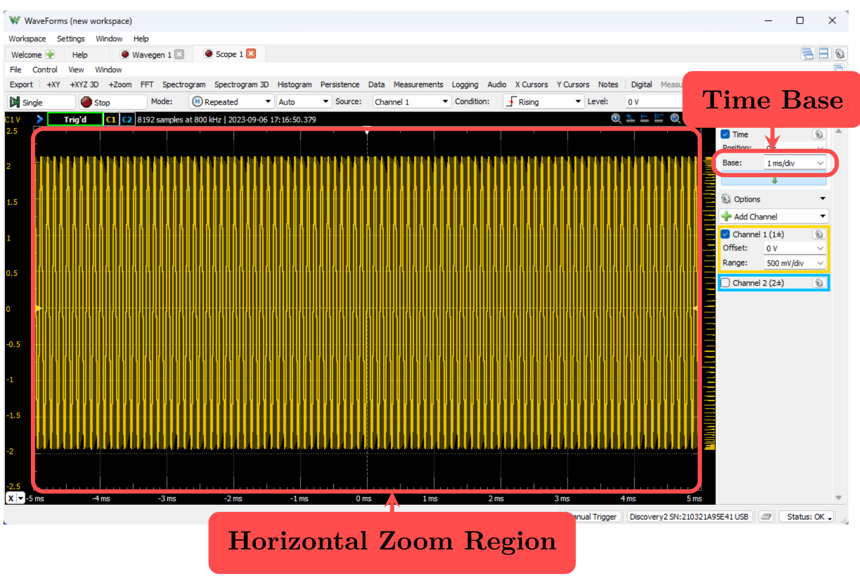


Figure 3.8: Scope with visible **CH. 1** signal. The **Time Base** control and **Horizontal Zoom Region** are highlighted.

Now, we will more deeply explore the DAD's oscilloscope. Adjust the horizontal zoom of

channel 1 (**CH. 1**). This can be accomplished either by scrolling the mouse wheel in the **Horizontal Zoom Region** or by editing the **Time Base** (see Fig. 3.8). Double-check to make sure the oscilloscope is stopped, then zoom in horizontally to a time base of 50 µs/div.

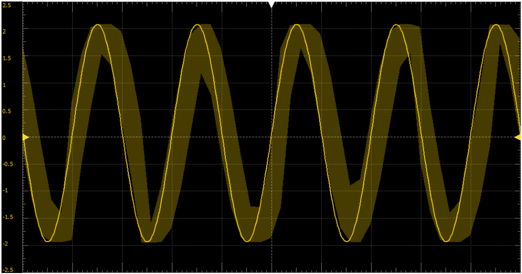
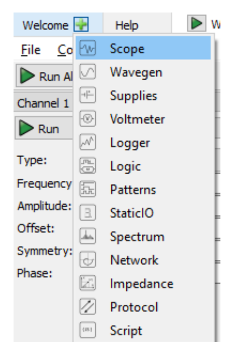


Figure 3.9: Scope for **Time Base** 50 µs/div. A "glow" surrounds the **CH. 1** signal.



Zooming horizontally reveals a translucent yellow glow around the **CH. 1** signal, as in Fig. 3.9. This results from stopping the **Scope** tab before zooming. This yellow translucent region represents the *precision error* for the current oscilloscope channel. The oscilloscopes present on the DAD change their sampling rates and precision depending on the *time base*. Hence, the *time base* (or 'horizontal zoom') isn't just some trick that WaveForms uses to scale (in software) data retrieved from a DAD oscilloscope; it's an actual setting that gets sent to the DAD, which determines, at a hardware level, the sampling rates and level of precision the DAD oscilloscope channels operate under. Press the **Run** button again. This will send the correct time base to the DAD, which will reconfigure itself to have better precision at this scale. The plot of **CH. 1** should become a lot clearer; see Fig. 3.10.

In addition to modifying the resolution (or 'zoom') of the horizontal (time) axis is possible to modify the amplitude resolution along the vertical axis. This can be done by either changing the **CH. 1 Range** or by moving the mouse into the **Vertical Zoom Region and scrolling** (see Fig. 3.10). Try modifying the vertical resolution to get a feel for how this works. Do not modify the horizontal resolution, keep it at 50 µs/div.

Press the **Stop** button on the **Scope** tab and zoom out to a *time base* of 200 µs/div. Some of the **CH. 1** plot seems to be missing; see Fig. 3.11. Based on your understanding of the DAD, explain why this is in your lab report.

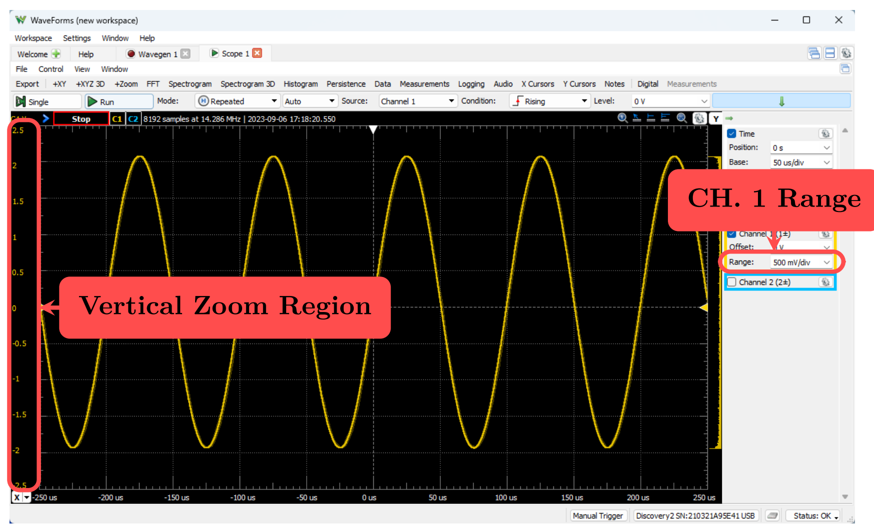


Figure 3.10: Scope with “glow” removed. The **CH. 1 Range** control and **Vertical Zoom Region** are highlighted.

Figure 3.11: Scope for **Time Base** 200 µs/div. The signal appears cut off.

3.4.1 The FFT View

Figure 3.12: Scope with **FFT View** visible. The **FFT Toggle** and **Drop Down** for the **FFT View** are highlighted.

Future exercises will discuss signals in terms of their *frequency* representation rather than *time*. An audio equalizer also decomposes signals in terms of some constituent frequencies and indicates the strengths of these frequencies. We will get a small taste of what a plot of this 'frequency domain decomposition' looks like.

Make sure the **Run** button in your **Scope** tab is pressed. Navigate to the **FFT View Toggle** (see Fig. 3.12) and click it to open the **FFT View** within the **Scope tab**. There is an additional **Drop Down** for the **FFT View** (see Fig. 3.12). Click the **Drop Down** to see the full array of controls for the **FFT View**.

Just as with the **CH. 1 plot**, the **Horizontal Zoom Region** encapsulates the viewable region of the **FFT plot** (see Fig. 3.13). Placing your cursor in this region and scrolling results in a modification of the horizontal resolution of the FFT. The same effect can be achieved by modifying the **Stop** control in the **FFT View** (see Fig. 3.13). Note that in this case, unlike the **CH. 1** plot, the horizontal axis represents *frequency*, not *time*.

The *time-domain* signal we are working with is a 10 kHz sinusoid. Ideally, the *frequency-domain* (FFT) representation should be mostly 'nothing' with an impulse-like spike at approximately 10 kHz. Yet, no matter how you change the horizontal scale amount in the **FFT View**,

Figure 3.13: **FFT View** enlarged along the horizontal axis. The **Stop** control and **Horizontal Zoom Region** for the **FFT View** are highlighted.

the lobe about 10 kHz doesn't get narrower as would be expected. To refine our resolution in the *frequency-domain*, we need more cycles of data in the *time-domain*. This property is called the *problem of conjugate variables* and is related to the *uncertainty principle*.[5] Don't worry about that for right now; just know that reducing horizontal resolution in the *time-domain* (i.e., making the *time base* larger) results in a finer resolution in the *frequency-domain*.

In the case of Fig. 3.13, the *time base* is 200 μs/div, changing it to 2 ms/div as in Fig. 3.14 gives a more impulse like spike in the frequency domain. The final point of order is adjusting the FFT units to afford a more intuitive representation vertically. Change the FFT units (see Fig. 3.14) to RMS($\tilde{\text{V}}$) to obtain a result more in-line with Fig. 3.15. Now, the *frequency-domain* representation should appear similar to an impulse at 10 kHz—which is what would be expected for a 10 kHz sinusoid.

Figure 3.14: **FFT View** with increased resolution. The **FFT Unit** control is highlighted.

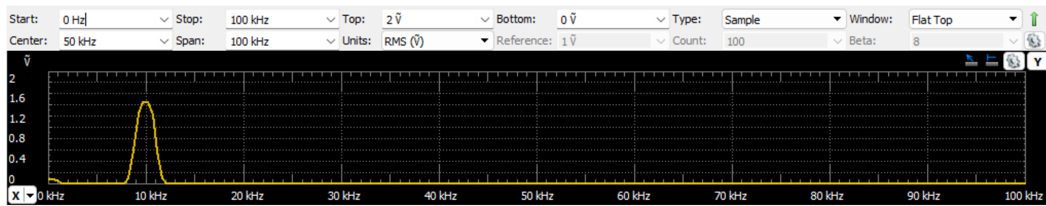


Figure 3.15: **FFT View** displayed with unit RMS($\tilde{\mathrm{V}}$).

3.4.2 Measuring Tools

Notice the toolbar at the top right of the **CH. 1** plot (see Fig. 3.16). It's really easy to miss but it contains a handy set of measuring tools to obtain various measurements for the signal visible in the viewing window. Let's focus on the first two tools:

: This button is the **L Tool**

: This button is the **H Tool**

The third/final tool is designed to measure pulse widths. None of the lab exercises make use of this tool.

The **L Tool** allows you to place three points in the shape of an 'L' while the **H Tool** does the same but in the shape of an 'H' (see Fig. 3.17). Both tools display the following measurements:

x_1: the x-position of the first point,

c_1: the y-position of the first point,

x_2: the x-position of the second point,

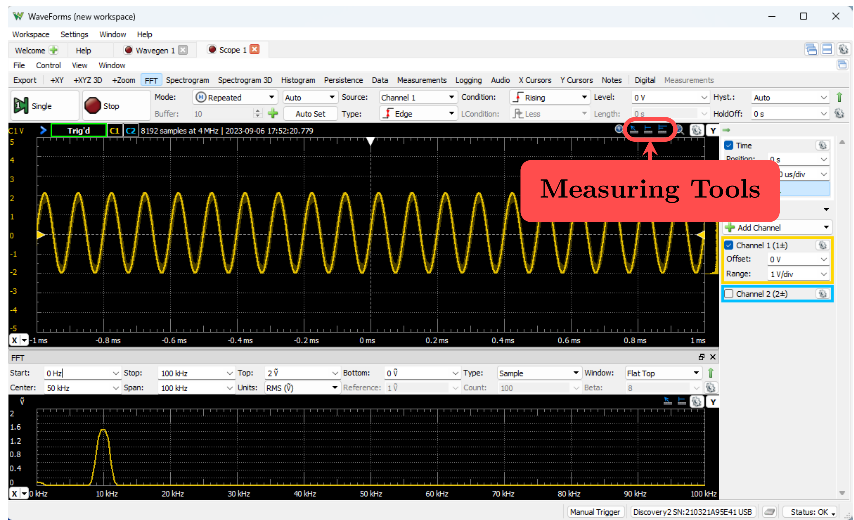
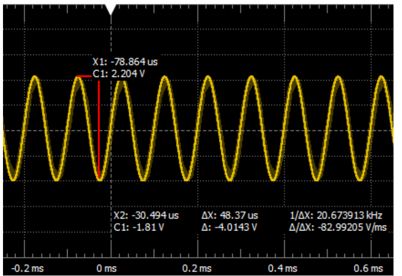


Figure 3.16: Scope with **Measuring Tools** highlighted.



(a) Scope with measurements produced by the **L Tool**.

(b) Scope with measurements produced by the **H Tool**.

Figure 3.17: The **L** and **H Tools**.

c_2: the y-position of the second point,

Δx: the distance $(x_2 - x_1)$ along the x-axis, and

Δ: the range $(c_2 - c_1)$ along the y-axis.

Right-clicking anywhere in the **Scope Viewing Regions** brings up the **Scope Menu**. This menu lets you change the *background color* and *add labels*. In your lab report, attempt to recreate the image in Fig. 3.19, complete with color scheme and label.

3.5 Capturing Data with the CyDAQ

Now that we've covered many of the basic features of the `WaveForms Software`, we'll move into some more advanced examples with the CyDAQ, such as using it as a filter, using it to capture

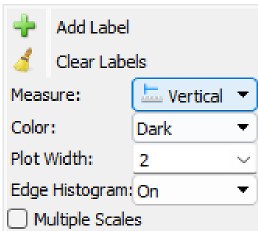


Figure 3.18: The **Scope Menu**.

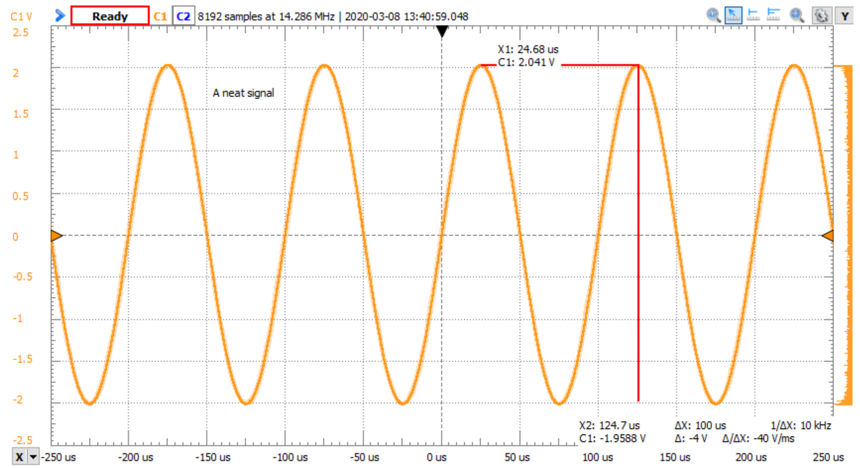


Figure 3.19: Image to reproduce.

data, and importing that data into MATLAB.

Firstly, in WaveForms, return to the **Wave Generator** (seen in Fig. 3.6) and enter the settings:

- **Type**: Square
- **Frequency**: 2 kHz
- **Amplitude**: 2 V
- **Offset**: 0 V
- **Symmetry**: 50%
- **Phase**: 0°

Make sure that the **Wave Generator** is running then switch back over to the **Scope** tab. Make sure that the scope is *running*. You should see something similar to Fig. 3.20. Adjust your **Time Base** and other controls if needed. Notice the harmonics (the additional bumps in the higher frequencies) present in the **FFT View** for the square wave.

Now return to the **Basic Settings Screen** of the `CyDAQ Software` (as seen in Fig. 3.4b).

Figure 3.20: A 2 kHz square-wave.

Now, we will configure the CyDAQ to act as a low-pass filter. Enter the following settings:

- **Sampling Rate**: 48000
- **Input**: Analog In
- **Filter**: 6th Order Low Pass
- **Mid Corner**: 2000

After entering these settings, click **Send Config** at the bottom of the **Basic Settings Screen**.

Switch back over to the Scope tab in *WaveForms*. You should see something similar to Fig. 3.21. Notice that the square wave has become a sinusoid. Also, notice that the harmonics that used to be present in the **FFT View** are now gone.

Switch back to the **Basic Settings Screen** of the `CyDAQ Software`. We are now going to capture some data. Click the **Start Sampling** button at the bottom of the screen and wait for sampling to begin. Once sampling has begun, quickly end sampling by clicking once more. Recall that the underlying signal has its lowest frequency at 2 kHz, which means there are 2000 periods a second. Hence, a tenth of a second contains plenty of information.

Once sampling/data capture has been completed, you will be prompted to save the data. Be sure to select to save the data as a `.mat` file. Create a new folder and save your data as `cydaq_square_filtered.mat`.

Open MATLAB and create a new script, `plot_filtered_square.m`, in the same folder as the `.mat` file. Both the script and the `.mat` file should be visible in MATLAB's **Current Folder**

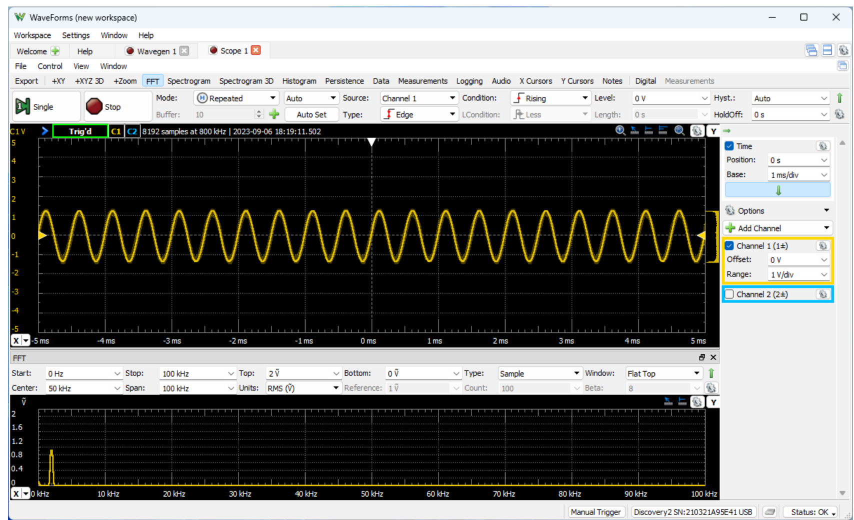


Figure 3.21: A 2 kHz square-wave filtered under a low-pass filter.

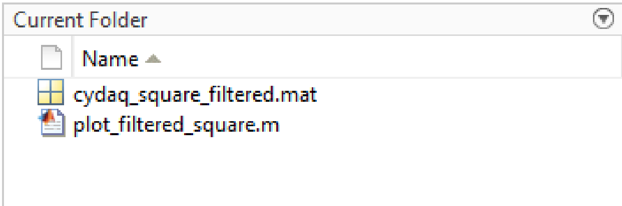


Figure 3.22: The **Current Folder View**.

View (see Fig. 3.22). Put the following contents into the script:

Listing 3.1: `plot_filtered_square.m`

```
1 clearvars; close all; clc;
```

The data we obtained was saved as a `.mat` file. A `.mat` file is a collection of MATLAB variables that will be copied into the **Variable Workspace** upon loading by the `load` function. We can view the variables stored within a `.mat` file by clicking on the `.mat` file within the **Current Folder View**. This will display the variables in the `.mat` file below the **Current Folder View** in an area called the **Current Contents View** (see Fig. 3.23).

The contents of the `.mat` file will contain a single variable, `data`. This variable is a two-column matrix with many rows. Add the following line to `plot_filtered_square.m`:

```
2 load('cydaq_square_filtered.mat')
```

This function, `load`, loads the `.mat` file (presuming it is in the same folder as the script) and

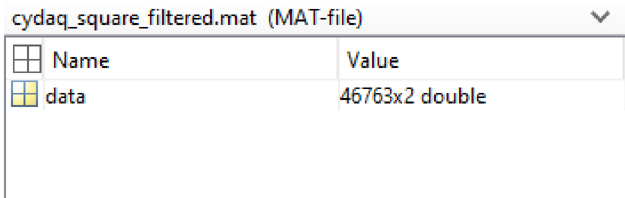


Figure 3.23: The **Current Contents View**.

places the variables residing within the `.mat` file into MATLAB's **Variable Workspace**. Run `cydaq_square_filtered.mat`. You should see that `data` is now in your **Variable Workspace**. Examine the contents of `data`. It should appear similar to Fig. 3.24.

data
46763x2 double

	1	2	3	4	5	6	7
1	2.2676e-05	-0.0696					
2	4.5351e-05	0.2821					
3	6.8027e-05	0.6020					
4	9.0703e-05	0.8681					
5	1.1338e-04	1.0366					
6	1.3605e-04	1.1148					
7	1.5873e-04	1.0928					
8	1.8141e-04	0.9780					
9	2.0408e-04	0.7582					

Figure 3.24: The contents of the `data` variable.

The way that the `CyDAQ Software` saves captured data is into a `.mat` file with the variable: `data`. The first column of `data` is the *time vector*. The second column of `data` is the signal data captured at the CyDAQ filter's output.

Using your knowledge of MATLAB so far, fill in the contents of `plot_filtered_square.m` such that it plots *approximately three periods* of the filtered signal, $y(t)$, with respect to the time vector t. Your plot should include a title and proper axis labels. Embed this plot within your lab report. Include your completed `plot_filtered_square.m` script as an attached file in your submission.

As a hint, your first attempt at plotting will probably appear very cluttered (similar to Fig. 3.25). This is because you've captured *way* more than three periods' worth of data. You likely have thousands or tens of thousands of periods captured. You'll need to use the `axis` command to zoom in on a region of the plot that shows approximately *three periods* of the

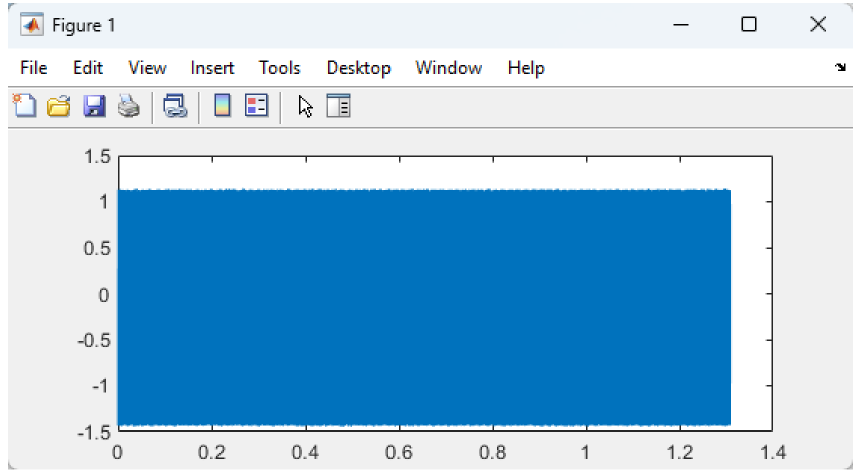


Figure 3.25: A cluttered MATLAB graph.

filtered square wave.

3.6 Practical Limitations of the CyDAQ

A frequently encountered task in electrical engineering is quickly transmitting a large amount of data. For instance, the USB 3.0 standard allows for data speeds of up to 5 Gbits/sec.[6] Digital signals encode bits (ones and zeros) at two distinct logic levels, usually (0 V and V_{cc}). Without thoughtful engineering, many wires and other components introduce *capacitance* at high frequencies, limiting data speeds. Consequentially, just because protocols like USB 3.0 allow for data speed of 5 Gbits, the physical properties of the cables or embedded peripherals may prevent this from being achieved. Often, there is a discrepancy between a system's theoretical maximum data speed (based on the clocking of internal components, etc.) and the achievable maximum data speed (due to capacitance in wires, electromagnetic interference, etc).

We will explore if such a discrepancy exists for the CyDAQ. The maximum sampling rate (and theoretical maximum data speed) of the CyDAQ is 48 kHz. Assuming a logic low corresponding to 0 V and a logic high corresponding to 5 V, the theoretical maximum data speed of 48 kHz implies that the CyDAQ should be able to oscillate between 0 V and 5 V at a rate of 24 kHz. However, the capacitance of the jumper wires and internal board traces renders this highly unlikely. We can characterize this capacitance and derive the actual, realizable data speed by sending a series of 0 V to 5 V pulses at well below half the maximum theoretical data speed (say

2 kHz) and measuring the rise-time. Generally, rise-time is measured as the time it takes for a signal to travel from 10% to 90% of its final value.

First, configure the CyDAQ as a pass-through. Return to the **Basic Settings Screen** of the `CyDAQ Software` (as depicted in Fig. 3.4b) and enter the settings:

- **Sampling Rate**: 48000 Hz
- **Input**: Analog In
- **Filter**: All Pass

Then click the **Send Config** button at the bottom of the **Basic Settings Screen**.

Go to the **Wave Generator** tab of the `WaveForms Software` and enter the settings:

- **Type**: Pulse
- **Frequency**: 2 kHz
- **Amplitude**: 5 V
- **Offset**: 0 V
- **Symmetry**: 50%
- **Phase**: 0°

Make sure that the **Wave Generator** is running then switch back over to the **Scope** tab.

The **Pulse** setting is very similar to the square except that the pulse starts at 0 V, whereas **Square** would've started at -2.5 V. Because of this, you will need to adjust the **Trigger Level** using either the control or the **Trigger Slider** (see Fig. 3.26). You may also need to adjust the **Offset** using either the control or the **Offset Slider**. The *trigger* controls at which y-intersection the time, $t = 0$ occurs (googling "oscilloscope trigger" might be helpful if you're confused).

In the **Scope View**, change your **Time Base** to something like 200 ns/div then use the measuring tools to measure the time it takes the signal to go from 0.5 V to 4.5 V and include a snapshot of your rise time measurement in your lab report. If your plot looks like it is low resolution and jittery, make sure your **Position** is set to 0 s in the horizontal settings (see Fig. 3.27).

3.7 Post-Lab Questions

For these questions, you may need to refer to the DAD specifications.[3]

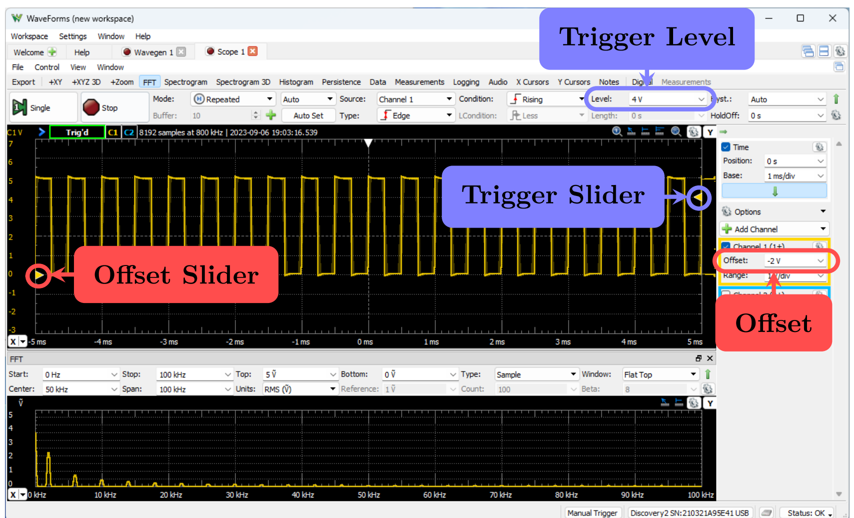


Figure 3.26: Scope with the **Offset** control, **Offset Slider**, **Trigger Level** control, and **Trigger Slider** highlighted.

Figure 3.27: Scope with the **Position** control, and **Position Slider** highlighted.

Q.1

What is the maximum AC amplitude of the function generator on the DAD?

Q.2

What is the maximum sampling rate of the oscilloscope (analog input) on the DAD?

3.8 Report Checklist

Work with lab partner(s) to create a lab report for this lab exercise. Be sure to include the names of each person in your group on the report. This report should be a typed document that outlines the basic procedure of the lab exercise and should include any results you were asked to obtain. Your report should follow a *cohesive narrative*—it should always be apparent *why* you have presented the results you have. Do not simply insert results without (1) motivating the problem and (2) providing an analysis of the data. Ensure that your report is grammatically correct with complete sentences. This does not mean your report must be wordy with filler—in fact, avoid filler and state your ideas concisely.

The final Post-Lab Questions section does need to be woven into your lab narrative. You may simply append your answers to each question at the end of your report in an attached answers section. Be sure to indicate which question you are answering by referencing the corresponding question number.

Be sure the following are included in your report:

- Section 3.4: Explain why stopping the **Scope** tab and increasing the time base results in missing **CH. 1** data, as in Fig. 3.11.
- Section 3.4.2: Recreate the graph in Fig. 3.19, complete with color-scheme, label, and measurements. Include an image of your recreation in your report.
- Section 3.5: Include the plot produced by your completed `plot_filtered_square.m` scrpit within your report.
- Section 3.5: Include your completed `plot_filtered_square.m` script as an attached file in your submission.
- Section 3.6: Include a snapshot of your rise time measurement in your report.
- Post-Lab Questions: Answer each question given.

3.9 Acknowledgments

This lab exercise is based on an exercise drafted by **Isaac Rex**.

3.R References

[1] "Zybo Z7 reference manual," Digilent. (2018), [Online]. Available: http://digilent.com/reference/programmable-logic/zybo-z7/reference-manual.

[2] "Analog discovery 2 reference manual," Digilent. (2018), [Online]. Available: http://digilent.com/reference/test-and-measurement/analog-discovery-2/reference-manual.

[3] "Analog discovery 2 specifications," Digilent. (2018), [Online]. Available: https://digilent.com/reference/test-and-measurement/analog-discovery-2/specifications.

[4] "Digilent waveforms," Digilent. (2024), [Online]. Available: https://digilent.com/shop/software/digilent-waveforms/.

[5] "Uncertainty principle," Wikipedia. (2023), [Online]. Available: https://en.wikipedia.org/w/index.php?title=Uncertainty_principle&oldid=1174833053.

[6] "USB 3.0," Wikipedia. (2024), [Online]. Available: https://en.wikipedia.org/w/index.php?title=USB_3.0&oldid=1224088325.

CyDAQ Impulse Responses

Characterizing Systems and Processing Signals With Impulse Responses

Overview

In this lab exercise, students will capture and utilize the impulse response of various systems. Students will configure the CyDAQ as a low-pass filter to filter the audio file, `noisy_audio.wav`, provided in this exercise's *supplementary materials*. Students will then capture the impulse response of the CyDAQ's low-pass filter using the WaveForms software. Students will complete `convolve_audio.m` (available in this exercise's *supplementary materials* and in Appendix 4.B) which is used to apply the same filtering to `noisy_audio.wav`—this time using MATLAB. Students will create a new script, `auralization.m`, which they will use to apply an impulse response file of their choice from the EchoTheif website to a recording of a human voice from the European Broadcasting Union's Sound Quality Assessment Material (SQAM). In addition to a typed report, students are asked to submit two completed MATLAB scripts, `convolve_audio.m` and `auralization.m`.

Learning Objectives

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

1. Relate the impulse response of an LTI system to its response response to any excitation.
2. Approximate an impulse in WaveForms.
3. Capture a system's response in WaveForms and export it to a csv file.
4. Use MATLAB to compute the response of an LTI system given an excitation signal and impulse response signal.

Materials

(1×)	DAD
(1×)	CyDAQ
(1×)	Hardware Speaker
(1×)	3.5 mm Male-to-Male Aux Cable
(1×)	3.5 mm Male-to-Breakout Aux Cable
(2×)	Female-to-Female Jumper Cables

4.1 Introduction

In this lab exercise, you will explore the impulse response which can be used to completely characterize an LTI system. That is, if you know the impulse response for an LTI system, you can compute its output for a given input. You will explore this in two ways.

First, you will use CyDAQ to clean up a noisy audio signal and measure the impulse response of the noise-removal system (CyDAQ). You will then use that impulse response in MATLAB to clean up the signal in the digital domain.

Next, You will explore an audio processing technique called auralization.[1] Auralization allows you to model the 'soundscape' of a room or space. For instance, you can model a giant music hall, a long hallway, a wide-open cathedral, etc. Then, you can use your model to process any audio sample and hear what that audio would have sounded like if it had been played in that room.

4.2 Characterizing Systems with Impulse Response

The impulse response function, $h(t)$, of a system, H, is defined as that system's response to a Dirac impulse, $\delta(t)$. For discrete-time systems, the impulse response, $h[n]$, is defined as that system's response to a Kronecker impulse, $\delta_K[n]$. The impulse response can completely characterize the behavior of *linear, time-invariant* (LTI) systems. In other words, if one knows a system's impulse response, one can compute that system's response (i.e., output) to *any* input.

For many first encountering this statement, this is very unintuitive yet astounding. It claims that one particular response of an LTI system, its impulse response $h(t)$, can be used to compute the response of that system, $y(t)$, for *any other input*, $x(t)$. Suppose you are given a black box system, in which all that is known is that the system is linear and time-invariant. If you fed an impulse function into the system and recorded its response, you could now compute the response of that black box system for *any input*. In some sense, you could say that an LTI system's impulse response *is* the system.

This black box thought experiment includes some important caveats. Firstly, it assumes the internal state of the system is zero. Secondly, it assumes that one can generate (i.e., physically realize) an input that is an impulse response. This isn't much of an issue for discrete-time systems, as Kronecker deltas are digitally realizable. However, Dirac deltas are not physically realizable and must be approximated. Thirdly, it assumes one could record the impulse response of a system. Impulse responses are not guaranteed to have an upper-time bound. In fact, any system with internal feedback will have an impulse response that has an infinite duration. Again, this may not present a major issue as stable systems have impulse responses that converge to zero—meaning that after a particular recording duration, one will have obtained a reasonable approximation of the impulse response. Nevertheless, this recording would still, technically, be an approximation. Even with these caveats, the underlying concept is still impressive: a single output of an LTI system (its impulse response) can be used to compute any output of the system. We could describe this process of obtaining an approximate impulse response of a system and using it to approximate the response to additional inputs as 'taking a snapshot of the system.'

The CyDAQ contains a built-in configurable digital LTI filter. This means these filters can be characterized by a discrete-time impulse response, $h[n]$. We can configure the CyDAQ to take input signals from a 3.5 mm Audio Jack, filter the audio signal, and send its output to an external speaker. Such a configuration is depicted in Fig. 4.1.

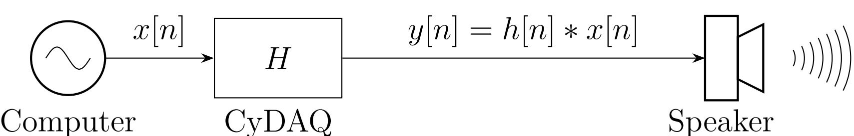


Figure 4.1: Block diagram for CyDAQ filter

To illustrate the process of 'taking a snapshot' of an LTI system and using it in place of the system to compute new responses, we will employ the configuration depicted in Fig. 4.1.

1. First, we will configure the CyDAQ to behave as a low-pass filter.
2. Next, we will demonstrate the ability of this low-pass filter to remove noise from a noise-corrupted audio file.
3. After this, we will approximate an impulse function and use it to retrieve an approximate impulse response of the CyDAQ's low-pass filter.
4. Finally, we will demonstrate that this impulse response serves as a suitable 'snapshot' of the CyDAQ's low-pass filter by using it to filter the same noise-corrupted file in MATLAB.

4.2.1 Playing Audio with the CyDAQ

To confirm the correct setup, we will first play the noise-corrupted audio file through the CyDAQ without filtering. The audio file `noisy_audio.wav` is available in this exercise's *supplementary materials*. You will need to connect the CyDAQ's input to your computer's headphone 3.5 mm Jack, and the External Speaker to your CyDAQ's output. You must also power the External Speaker using the CyDAQ's 5 V sensor terminal. This is accomplished using the following connections:

- Connect the **PC's Headphone Terminal** to the **CyDAQ's 3.5 mm Audio In Terminal** (`J8`), using a **3.5 mm Male-to-Male Aux Cable**.
- Connect the **CyDAQ's V_{out} Terminals** (`J10`) to the **Speaker's 3.5 mm Audio In Terminal**, using the **3.5 mm Male-to-Breakout Aux Cable**. You will need to use **Female-to-Female Jumper Cables** to connect the **Male Breakout Terminals** to the **Male V_{out} Terminals**.
  - Connect the **Red Breakout Cable** to the **Positive (+) V_{out} Terminal** (`Vout+`).
  - Connect the **Black Breakout Cable** to the **Negative (-) V_{out} Terminal** (`Vout-`).
- Connect the **Speaker's Power Connector** to the **CyDAQ's 5 V Sensor In Terminal** (`J11`) using the **Black/White Wires** attached to the speaker.

Refer to Figs. 4.3a and 4.3b in Appendix 4.A for locating the proper terminals.

Next, open the `CyDAQ Software` and send the following configuration:

- **Sampling Rate**: 48 000 Hz
- **Input**: Audio In
- **Filter**: All Pass

Note that the input is set to **AUDIO In**. Be sure to run the CyDAQ configuration **after** all the connections have been made. Setting the filter type to “All Pass” is equivalent to $h(t) = \delta(t)$. Using any media player of your choice, play `noisy_audio.wav`. Make sure that your audio output device is properly configured to play audio out the headphone jack. The External Speaker has much less power than typical PC speakers. Hence, you should turn up your system’s volume to the maximum to be able to hear sound from the External speaker. If everything has been set up correctly, you should hear a short piano piece play from the external speaker. There should be noticeable high-frequency interference present in the audio clip.

4.2.2 Filtering with the CyDAQ

Now that you have confirmed that the CyDAQ has been properly configured to play audio, we will configure a low-pass filter to remove the high-frequency interference present in `noisy_audio.wav`. In the `CyDAQ Software`, send the following parameters:

- **Sampling Rate**: 48 000 Hz
- **Input**: Audio In
- **Filter**: 6th Order Low Pass
- **Mid Corner:** 6000 Hz

Play the `noisy_audio.wav` once again from your computer. You should notice that the high-frequency interference is no longer perceivable; it has been effectively removed.

4.2.3 Measuring the CyDAQ’s Impulse Response

Now that we’ve used the CyDAQ to filter a noisy audio file, we will recreate this process by taking a ‘snapshot’ of the CyDAQ’s low-pass filter and applying it to the noisy audio file in MATLAB. We will do this by recording the (approximate) impulse response of the CyDAQ’s low-pass filter and convolving it with the signal stored in `noisy_audio.wav`. Recall from Sec. 4.2 that impulse responses are typically challenging to realize. Though the CyDAQ is a digital

system, there is a difference in clock speeds between the DAD and the CyDAQ as well as the CyDAQ and the configurable filter that make it difficult to realize an exact Kronecker impulse response. Instead, we will approximate it by generating a narrow pulse on the DAD.

It should be noted that electrical system impulse responses are not typically measured this way. Using a chirp[2] or a frequency sweep to measure the system response is much more common. Mechanical systems, however, are sometimes measured with an approximated impulse. A special hammer[3] is used to whack the system and measure the resulting vibrations. For our purposes, due to the simplicity of the filter, a narrow pulse proves to be an adequate substitute for impulse.

We will now reconnect the CyDAQ to receive the impulse approximation from the DAD. Disconnect the 3.5 mm Male-to-Male Aux Cable from the CyDAQ. Disconnect the 3.5 mm Male-to-Male Aux Cable from your PC. Completely disconnect all connections between the External Speaker and the CyDAQ.

- Connect the **Waveform Generator 1** (`W1`) and **Ground** (`GND`) **Jumper Wires** on the DAD to the **CyDAQ's V_{in} Terminals**, (`Vin+`) and (`Vin-`), respectively.
- Connect **Scope Ch. 1 Negative** (`CH1-`) and **Scope Ch. 1 Positive** (`CH1+`) on the DAD to the **CyDAQ's V_{out} Terminals**, (`Vout-`) and (`Vout+`), respectively.

Now the CyDAQ has been wired to receive input from the DAD's WaveGen and transmit output to the DAD's oscilloscope. However, we still need to tell the CyDAQ to accept input from its V_{in} terminal rather than its 3.5 mm Audio terminal. Open the `CyDAQ Software` and send the following configuration:

- **Sampling Rate**: 48 000 Hz
- **Input**: Analog In
- **Filter**: 6th Order Low Pass
- **Mid Corner:** 6000 Hz

Note that the input is set to **ANALOG In**. Next, we will configure the DAD's WaveGen to output our approximate impulse function. Launch the `WaveForms Software` and open a new **Wavegen** tab. Enter the following configuration:

- **Type**: Pulse
- **Frequency**: 100 Hz
- **Amplitude**: 5 V

- **Offset**: 0 V
- **Symmetry**: 0.2%
- **Phase**: 0°

Press the **Run** button on the **Scope** tab in WaveForms.

Adjust the oscilloscope in WaveForms to clearly capture the response. It should look similar to Fig. 4.2. Once you've finished adjusting the scope, **Stop** the oscilloscope. Include a screenshot of the measured impulse response in your report. Note that the filter is physically implemented with an *inverting amplifier*, which has a negative gain, so you are actually measuring $-h(t)$. This will not affect any of this experiment, but if you wish, you may multiply the impulse response by -1 in MATLAB to get $h(t)$.

Now, we will export the captured waveform as a `.csv` so you can load it into MATLAB. Select the **Export** button (just below the File/Control/View menu ribbon) in Waveforms to launch the **Export Dialog**. Uncheck the comments box, but leave the **Headers** and **Labels** boxes checked. Create a new folder on your PC to serve as your project directory and save the captured impulse response as `impulse_response.csv` in that folder.

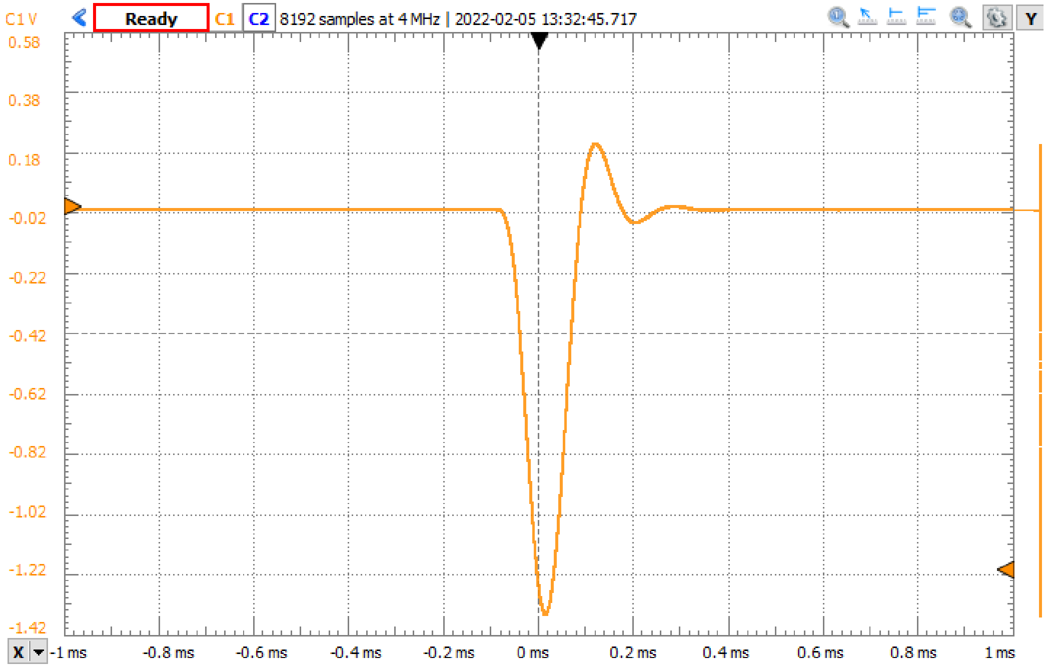


Figure 4.2: Impulse response of CyDAQ measured on the DAD

4.2.4 Using the Impulse Response as a Filter

Now that you have captured the (approximate) impulse response of the CyDAQ system, we can use it to filter any input signal digitally. As before, we will choose our input to be `noisy_audio.wav`. Move a copy of `noisy_audio.wav` into your project directory.

The skeleton of a MATLAB script that applies a captured impulse response (stored in a `.csv` file) to an audio file has been provided to you as `convolve_audio.m` in this exercise's *supplementary materials* as well as in Appendix 4.B. Copy `convolve_audio.m` into your project directory and open it in MATLAB. You will need to complete the portions of this file containing `TODO`. Note the portion of this code that calls the `resample(...)` function. To properly convolve the captured impulse response with the noisy audio, the sampling rates of both signals must be the same. However, the sampling rate of the DAD (which captured the impulse response) is much faster than the sampling rate of the audio file. Consequentially, there is some code in `convolve_audio.m` which resamples the impulse response from the `.csv` file, `h_old`, to have the same sampling rate as the audio file.

Once you've filled in each of the `TODO` portions of `convolve_audio.m`, run the script. If done correctly, you should see two plots, one of the (resampled) captured impulse response and another of the noisy and filtered audio signals overlaid atop one another. You should also hear the original `noisy_audio.wav` play followed by the filtered version. You may need to disconnect the 3.5 mm Aux cable from your computer if you forgot to do so.

Once you've confirmed that you've correctly filled in `convolve_audio.m`, include both plots it produces in your lab report. Include a copy of your completed `convolve_audio.m` as an attached file in your submission. Recall how the filtered audio sounded when you used the physical CyDAQ as a filter. Does the result sound much different from your implementation in MATLAB? Discuss this in your report.

4.3 Auralization

The next topic covered in this lab exercise is auralization. Just as impulse responses let us take 'snapshots' of LTI electrical systems (like filters), auralization allows us to take 'snapshots' of LTI

sonic/acoustic systems like echo and reverberation. In essence, if someone records the impulse response of a large hall, cathedral, cave, etc., it is possible to simulate other sounds playing in that area. EchoThief[4] is a website that collects interesting impulse responses from "unusually clamorous places" and hosts them free for anyone to use. You will use some of these impulse responses to make a sound clip sound as though it was recorded in the chosen environment. Navigate to the EchoThief website[4] and select an impulse response. It should download as a `.wav` file. Save the `.wav` file containing your EchoThief impulse response in your project folder.

We will also need a new sound to use as an input to apply echo/reverb. The human voice is a great candidate for this task. High-quality recordings of human speech are available for educational use as part of the European Broadcasting Union's *Sound Quality Assessment Material* (SQAM) package.[5] The Massachusetts Institute of Technology website hosts several of these SQAM recordings. Navigate to the Massachusetts Institute of Technology's SQAM webpage[6] and download the two English speech samples: `spfe49_1.wav` (female speech) and `spme50_1.wav` (male speech). Save `spfe49_1.wav` and `spme50_1.wav` in your project folder.

Create a new MATLAB script called `auralization.m` in your project folder. Since the process of applying a captured impulse response to an audio file is similar to your task in Sec. 4.2.4, we can reuse much of the `convolve_audio.m`. Copy the contents of `convolve_audio.m` into `auralization.m`.

Now we will need to modify the script to work properly for our auralization task. First, change `audio_sample_file` to point to either `spfe49_1.wav` and `spme50_1.wav`. Next, change `impulse_response_file` to point the `.wav` file you downloaded from EchoTheif.

Working our way through the script, we see that line 17, which loads the input audio file, still works properly for loading our human speech `.wav` files. Moving on to line 25, we see that the impulse response data is loaded using a `readmatrix(...)` function. This function loads `.csv` files and will no longer work for us since our impulse response function from EchoTheif is stored in a `.wav` file. Delete lines 25–32 (everything before the resampling command).

In their place we'll need to insert a few lines of code to properly load the EchoTheif impulse response. First, we'll need to load the impulse response data and sampling rate into `H` and `Fs_h`, respectively:

```
[H, Fs_h] = audioread(impulse_response_file);
```

Next, we'll need to pull out the first channel of the audio file and store it in the variable `h_old`:

```
h_old = H(:, 1); % Grab the first (left) channel
```

Finally, the `resample(...)` function later in the script requires a time vector, `t_h_old`, that corresponds to the signal in `h_old`:

```
th_old = (0:length(h_old)-1)./Fs_h;
```

With these changes, your `auralization.m` script should now properly apply the EchoTheif impulse response to the human speech audio file.

Run `auralization.m` and listen to the result. Discuss what you hear in your lab report. Include both plots produced by `auralization.m`. Include a copy of your completed `auralization.m` as an attached file in your submission.

4.4 Report Checklist

Work with lab partner(s) to create a lab report for this lab exercise. Be sure to include the names of each person in your group on the report. This report should be a typed document that outlines the basic procedure of the lab exercise and should include any results you were asked to obtain. Your report should follow a *cohesive narrative*—it should always be apparent *why* you have presented the results you have. Do not simply insert results without (1) motivating the problem and (2) providing an analysis of the data. Ensure that your report is grammatically correct with complete sentences. This does not mean your report must be wordy with filler—in fact, avoid filler and state your ideas concisely.

Be sure the following are included in your report:

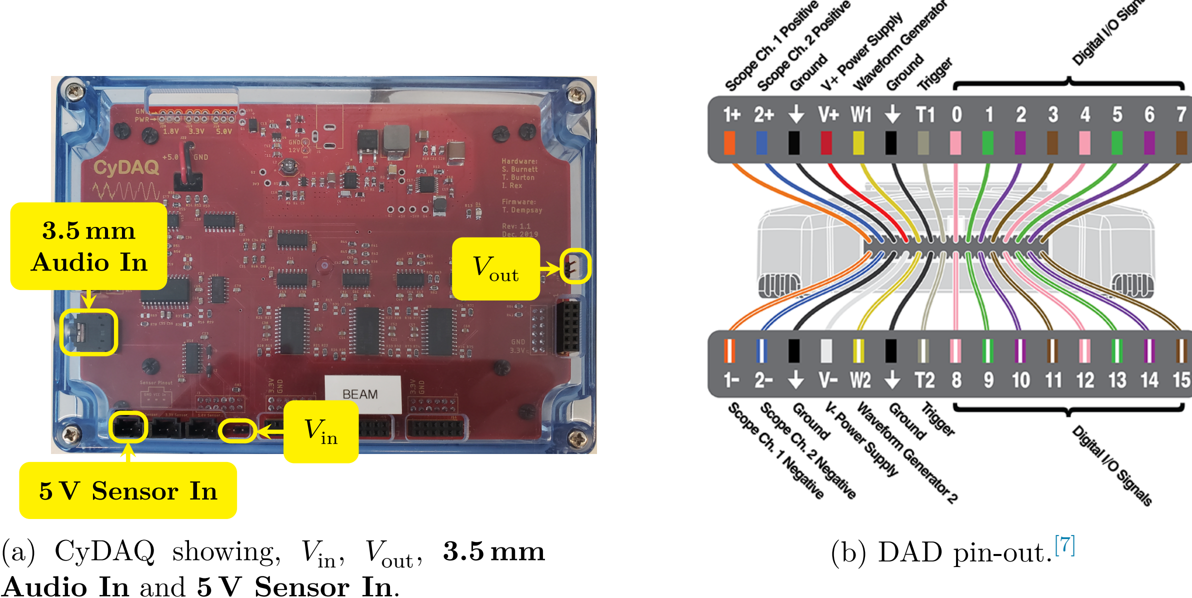
- Section 4.2.3: Include the scope capture of the system's measured impulse response.
- Section 4.2.4: Include the plot of the measured impulse response produced by `convolve_audio.m`.
- Section 4.2.4: Include the plot of the noisy and filtered audio signals produced by `convolve_audio.m`.
- Section 4.2.4: Include the completed copy of `convolve_audio.m` as an attached file in your submission.

- Section 4.2.4: Provide some discussion comparing results of filtering using the physical filter on the CyDAQ versus the impulse response 'snapshot' in MATLAB.
- Section 4.3: Discuss the effect that the EchoTheif impulse response had when applied to your human speech file.
- Section 4.3: Include the plot of the EchoTheif impulse response produced by `auralization.m`.
- Section 4.3: Include the plot of the original and filtered audio signals produced by `auralization.m`.
- Section 4.3: Include the completed copy of `auralization.m` as an attached file in your submission.

4.5 Acknowledgments

This lab exercise is based on an exercise drafted by **Isaac Rex**.

4.A Hardware Ports



(a) CyDAQ showing, V_{in}, V_{out}, **3.5 mm Audio In** and **5 V Sensor In**.

(b) DAD pin-out.[7]

Figure 4.3: Pin-outs/terminals for CyDAQ and DAD.

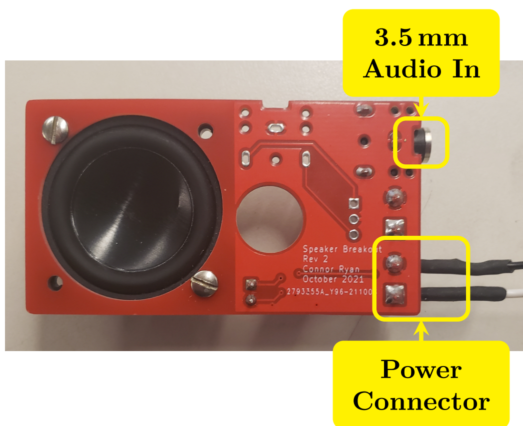


Figure 4.4: CyDAQ Speaker showing **3.5 mm Audio** terminal and **Power Connector**.

4.B Convolve Audio MATLAB Script

Listing 4.1: convolve_audio.m

```
clearvars; close all; clc;
% Audio Convolution Excercise
% Written By: Isaac Rex, 2020
% Updated By: Aaron Fonseca, 2024

% **Fill-in all TODOs**

% **Make sure that all the files are in the same folder as your MATLAB
% working directory**

impulse_response_file = 'TODO'; % TODO: fill in the filename of your csv
audio_sample_file  = 'noisy_audio.wav';

% =====================================================
% Load in the impulse response and audio sample data
% =====================================================
[X, Fs] = audioread(audio_sample_file); % Load the audio sample
x = X(:, 1); % Grab the first (left) channel

% The lines below load the measured impulse response. Because the data is
% being processed on a computer, continous time is being approximated by a
% a discrete time system. In order to sync up the time t, with the discrete
% index n, the sampling rate needs to be adjusted.

H = readmatrix(impulse_response_file); % Load the impulse response file
h_old = H(:, 2);                   % Amplitude values of h
t_h_old = H(:, 1);                 % Time vector corresponding to h_old

% Compute Fs_h
dt_h = H(2, 1) - H(1, 1);          % Get the sample period of the
                                   % impulse response.
```

```
32 Fs_h = round(1/dt_h);            % Get the sampling frequency.
33
34 % Resample h from old sampling rate, Fs_h, to new sampling rate, Fs
35 [h, ] = resample(h_old, t_h_old, Fs); % Match the sampling rate of impulse
36                                   % response to audio sample.
37 t_h = (0:length(h)-1)./Fs;        % Recalculate the time vector for
38                                   % the impulse response, h.
39
40 % =====================================================
41 % Plot the resampled impulse response
42 % =====================================================
43 figure(1);
44 clf;
45 plot(t_h, h, 'LineWidth', 2);
46
47 grid on;
48 set(gca, 'FontSize', 18)
49 xlabel('TODO', 'FontSize', 22)
50 ylabel('TODO', 'FontSize', 22)
51
52 % =====================================================
53 % Convolve the audio signal with the impulse response
54 % =====================================================
55 % Use the 'same' argument in conv() so that y is the same length as x.
56 % Type "help conv" in the console for details. Make sure to use the
57 % 'same' option!
58 y = TODO; % fill in with conv
59
60 % Now we need to 'normalize' y. The audioplayer in MATLAB expects y to be
61 % in the range [-1, 1], so it needs to be scaled to match that range.
62 y = y ./ (max(abs(y)));
63
64 % =====================================================
65 % Play the sounds
66 % =====================================================
67 ap_x = audioplayer(x, Fs);
```

```
68 ap_y = audioplayer(y, Fs);
69
70 playblocking(ap_x);
71 playblocking(ap_y);
72
73 % =====================================================
74 % Plot the noisy and filtered audio clips on the same plot
75 % =====================================================
76 t = (0:length(x)-1)./Fs; % Generate time vector
77
78 % plot
79 figure(2)
80 clf;
81 plot(TODO, TODO, 'LineWidth', 1);
82 hold on;
83 plot(TODO, TODO, 'LineWidth', 1);
84
85 grid on;
86 set(gca, 'FontSize', 18)
87 xlabel('TODO', 'FontSize', 22)
88 ylabel('TODO', 'FontSize', 22)
89 legend('TODO', 'TODO')
```

4.R References

[1] "Auralization," Wikipedia. (2023), [Online]. Available: http://en.wikipedia.org/w/index.php?title=Auralization&oldid=1119847521.

[2] "Chirp," Wikipedia. (2023), [Online]. Available: http://en.wikipedia.org/w/index.php?title=Chirp&oldid=1174787830.

[3] "Impulse force hammer," Kistler. (2023), [Online]. Available: http://www.kistler.com/US/en/impulse-force-hammer/C00000121.

[4] C. Warren. "Echothief." (2023), [Online]. Available: http://www.echothief.com/.

[5] European Broadcasting Union. "Sound quality assessment material recordings for subjective tests." (Oct. 2008), [Online]. Available: https://tech.ebu.ch/publications/sqamcd.

[6] European Broadcasting Union. "SQAM - sound quality assessment material." (Jun. 2001), [Online]. Available: https://sound.media.mit.edu/resources/mpeg4/audio/sqam/.

[7] "Analog discovery 2 reference manual," Digilent. (2018), [Online]. Available: http://digilent.com/reference/test-and-measurement/analog-discovery-2/reference-manual.

Frequency Response and System Identification

Measuring the Frequency Response of Systems

Overview

In this lab, students will use the DAD to measure the frequency response of a 'mystery system' programmed onto the CyDAQ by the *Electronics and Technology Group* (ETG). Students will compute the frequency response using two separate approaches. First, students will conduct a series of manual measurements using the WaveForms software. Next, students will use the Loopinator software to automate the process using a frequency sweep. Finally, students will use MATLAB to produce plots comparing their manual measurements to the results obtained using the Loopinator software.

Learning Objectives

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

1. Use the WaveForms software to measure the gain of a sinusoidal excitation.
2. Use the WaveForms software to compute the phase difference of a sinusoidal excitation.
3. Use the Loopinator software to compute the frequency response of a system and export it into MATLAB.
4. Convert measurements of gain into decibels.

Materials

(1×) DAD
(1×) CyDAQ
(1×) Jumper Splitter

5.1 Introduction

In this lab exercise, we will use the DAD to measure the frequency response of an unknown system that has been loaded onto the CyDAQ by Iowa State University's *Electronics and Technology Group* (ETG). Frequency response measurement is a powerful tool for characterizing *linear, time-invariant* (LTI) systems and predicting how they will respond to future excitations. The particular frequency response methodology employed by this exercise involves conducting a series of manual measurements of the responses to individual sinusoids across a range of frequencies. It relies on a mathematical relationship that relates the time-domain properties of a sinusoidal excitation to the system's frequency response function. After collecting manual measurements, we will use the `Loopinator Software` to automate the process and measure the system's frequency response via a frequency sweep. The results from the `Loopinator Software` will then be exported into MATLAB and used as a comparison against the manual measurements we collected earlier.

5.2 Background on Frequency Response

There are several ways to measure the frequency response of a system, but one of the most common (and hopefully most intuitive) is to simply apply a sine wave to the input and measure the response in terms of the *gain* and *phase delay*. Sinusoidal excitations are somewhat unique in that they have a rather direct relationship between the time-domain expression and the frequency response function. This relationship is illustrated in Fig. 5.1.

$$\cos(\omega t) \longrightarrow \boxed{H(j\omega)} \longrightarrow |H(j\omega)| \cos(\omega t + \angle H(j\omega))$$

Figure 5.1: Block diagram illustrating the effects of an LTI system on a sinusoidal excitation.

From Fig. 5.1, we can see that the magnitude of the frequency response $\big|H(j\omega)\big|$ is the ratio between the final amplitude of the sinusoid and the initial amplitude; in other words, the *gain*. We can also see that the phase of the frequency response $\angle H(j\omega)$ is the *phase difference* between the output and the input.

Recall that $H(j\omega)$ is a complex number and that the magnitude, $\big|H(j\omega)\big|$, and phase, $\angle H(j\omega)$,

is the expression of this number in polar form. Typically, graphing $H(j\omega)$ entails producing the two separate plots, one of the $|H(j\omega)|$ and one of $\angle H(j\omega)$. Both are plotted with respect to frequency, ω, across the same range of values (i.e., share the same ω-axis) with a *logarithmic spacing*. Sometimes, rather than angular frequency, ω, the plots are produced with respect to Hertzian frequency f (recall that $f = 2\pi\omega$). Often, the magnitude is plotted in decibels because it affords a larger dynamic range. Conversion of $|H(j\omega)|$ to dB is using the equation:

$$|H(j\omega)|_{\text{dB}} = 20\log_{10}\left(|H(j\omega)|\right). \tag{5.1}$$

5.3 System Identification using Frequency Response

We will now use our knowledge of frequency response to identify/characterize a 'mystery' system that has been loaded onto the CyDAQ by ETG. Since you need to connect both the Wavegen and the Scope to the input of CyDAQ, you will need to attach a Jumper Splitter to the CyDAQ. Make the following connections:

- Connect the **Jumper Splitter** to the **CyDAQ's V_{in} Terminals**, (`Vin-`) and (Vin+).
- Connect the **DAD's Ground** (`GND`) and **Scope Ch. 1 Negative** (`CH1-`) **Terminals** to the **CyDAQ's Negative V_{in} Terminal** (`Vin-`), on the **Splitter**.
- Connect the **DAD's Waveform Generator 1** (`W1`) and **Scope Ch. 1 Positive** (`CH1+`) **Terminals** to the **CyDAQ's Positive V_{in} Terminal** (`Vin+`), on the **Splitter**.
- Connect the **DAD's Scope Ch. 2 Negative** (`CH2-`) and **Scope Ch. 2 Positive** (`CH2+`) **Terminals** to the **CyDAQ's V_{out} Terminals**, (`Vout-`) and (`Vout+`), respectively.

The pinouts for the CyDAQ and the DAD are given in Fig. 5.6 in Appendix 5.A. Launch the `WaveForms Software` and open a new **Wavegen** tab. Enter the following configuration:

- **Type**: Sine
- **Frequency**: 100 Hz
- **Amplitude**: 5 V
- **Offset**: 0 V
- **Symmetry**: 50%
- **Phase**: 0°

Once this configuration is entered, click the **Run** button.

5.3.1 Manual Measurements in WaveForms

Measuring Gain in WaveForms

Launch a **Scope** tab in WaveForms. Ensure you have both Channels enabled and that both are visible in the **Scope** viewport. Recall that a system's gain, G, is the ratio of the output amplitude over the input amplitude:

$$|H(j\omega)| = G = V_{\text{out}} \,/\, V_{\text{in}}. \tag{5.2}$$

Like standalone hardware oscilloscopes, the WaveForms scope comes equipped with a handy feature: *measurements*. Open the **Measurements Side View** by clicking the **Measurements** button on the upper command ribbon. The location of the **Measurements** button is illustrated in Fig. 5.2.

Once you've opened the **Measurements Side View**, click the **Add** button in the side view and select **Defined Measurement**. A new dialog window will open that consists of two columns of expandable lists. The left column will contain a list of oscilloscopes channels. Select Channel 1 for now. The right column contains all the available measurements for the currently selected channel. We want to measure the signal amplitude —this is a *vertical* measurement. Expand the list node labeled **Vertical** and select **Amplitude** from the child list. Click **Add** to add the amplitude *measurement* to the **Measurements Side View**. Repeat this process for Channel 2.

Once you've added amplitude measurements for both channels, you can **Stop** the scope and record their values. From the connections we made in Sec. 5.3, we know that Channel 1 corresponds to our input signal and Channel 2 corresponds to our output signal. We can then use the amplitude measurements in WaveForms to compute the gain at the current frequency via Eq. (5.2).

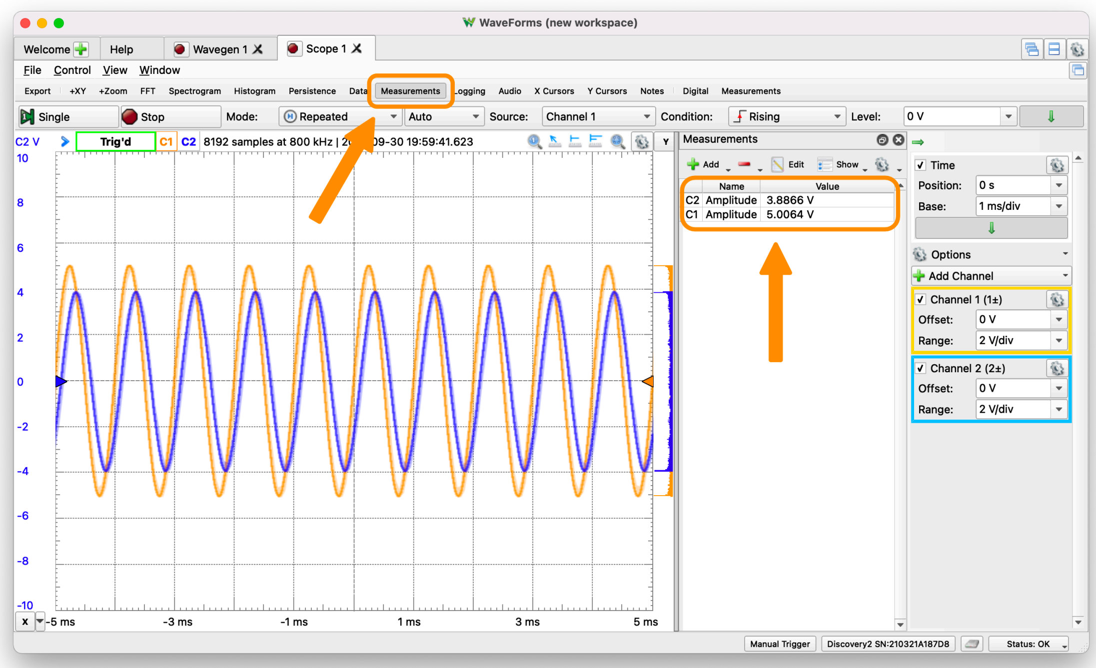


Figure 5.2: Measuring amplitude with **Measurements** feature

Measuring Phase in WaveForms

Measuring the phase is slightly more involved–there is no phase-difference *measurement*. Instead, you'll have to measure the time delay from the input signal to the output signal using either the L-tool or the H-tool (these are discussed in the Introduction to CyDAQ exercise) then convert that time delay to a phase. The formula to convert a time delay to a phase is:

$$\angle H(j\omega) = \phi^\circ = 360^\circ \cdot f \cdot \Delta t. \tag{5.3}$$

where f is the signal frequency in Hz and Δt is the time delay in seconds. In your report, explain why Eq. (5.3) is the conversion formula. I.e., explain how you might come up with the formula if you couldn't remember it. *Hint*: Think of $f \cdot \Delta t = \frac{1}{T}\Delta t$ as the fraction or percent of a period by which the signal is delayed.

Fig. 5.3 shows an *example* of measuring the phase delay. This is only an example and may not match your values. You may need to change your time base to get an accurate measurement. In this example, the frequency of the signal is 1 kHz, so $\phi^\circ = 360^\circ \cdot 1\,\text{kHz} \cdot 109\,\mu\text{s} = 39.2^\circ$.

When measuring the time delay for signals with small amplitudes (sub 100 mV), the *DC offset* from the op-amps inside the CyDAQ can cause your Channel 2 voltage to drift above or below the `GND` reference. You may need to adjust the channel offset to account for this. When measuring, ensure that you measure between the same points on the two sine waves (e.g., always measure from peak to peak or from zero-crossing to zero-crossing as is done in Fig. 5.3).

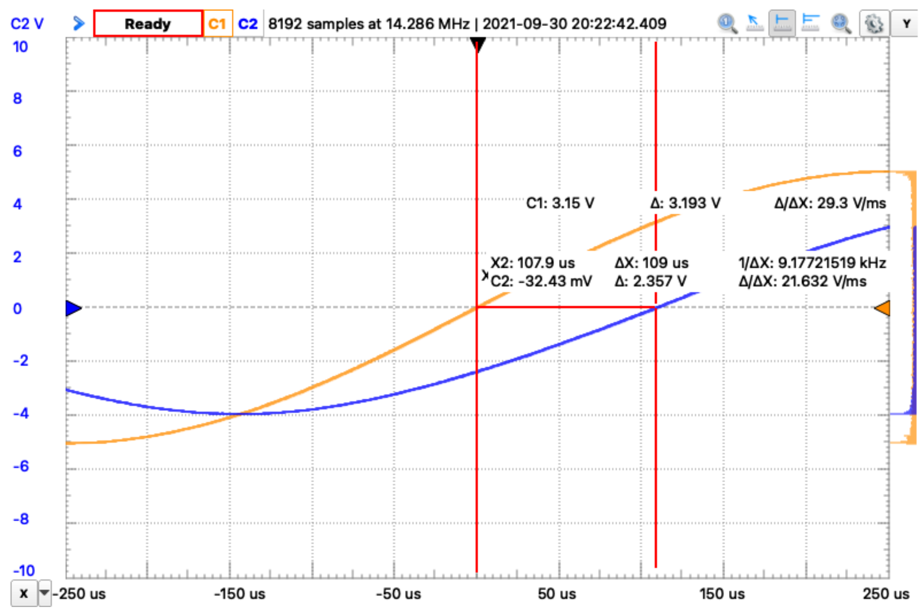


Figure 5.3: Measuring time delay of output (blue). Phase delay $\phi^{\circ} = 360^{\circ} \cdot f \cdot \Delta t$

5.3.2 Tabulate Your Results

Frequency	Gain [Ratio]	ϕ [Degrees]
100 Hz		
1 kHz		
10 kHz		
100 kHz		

Table 5.1: Table of manual measurements

Repeat the steps in Secs. 5.3.1 and 5.3.1 for each frequency value in Table 5.1. Include the completed table in your lab report. Discuss these results. Based on your measurements, what type of filter do you believe has been deployed on the CyDAQ.

5.3.3 Automated Measurements

Now that you've completed a series of manual measurements, we will compare your results to the frequency response calculated using an automated method. Close the `WaveForms Software`. The frequency response estimation software we will use is called `Loopinator`. It is available on each of the lab computers courtesy of the *Electronics and Technology Group* (ETG). It can be located by searching "Loopinator" in the Windows Start Menu. Launch Loopinator. Fig 5.4 depicts the main Loopinator interface.

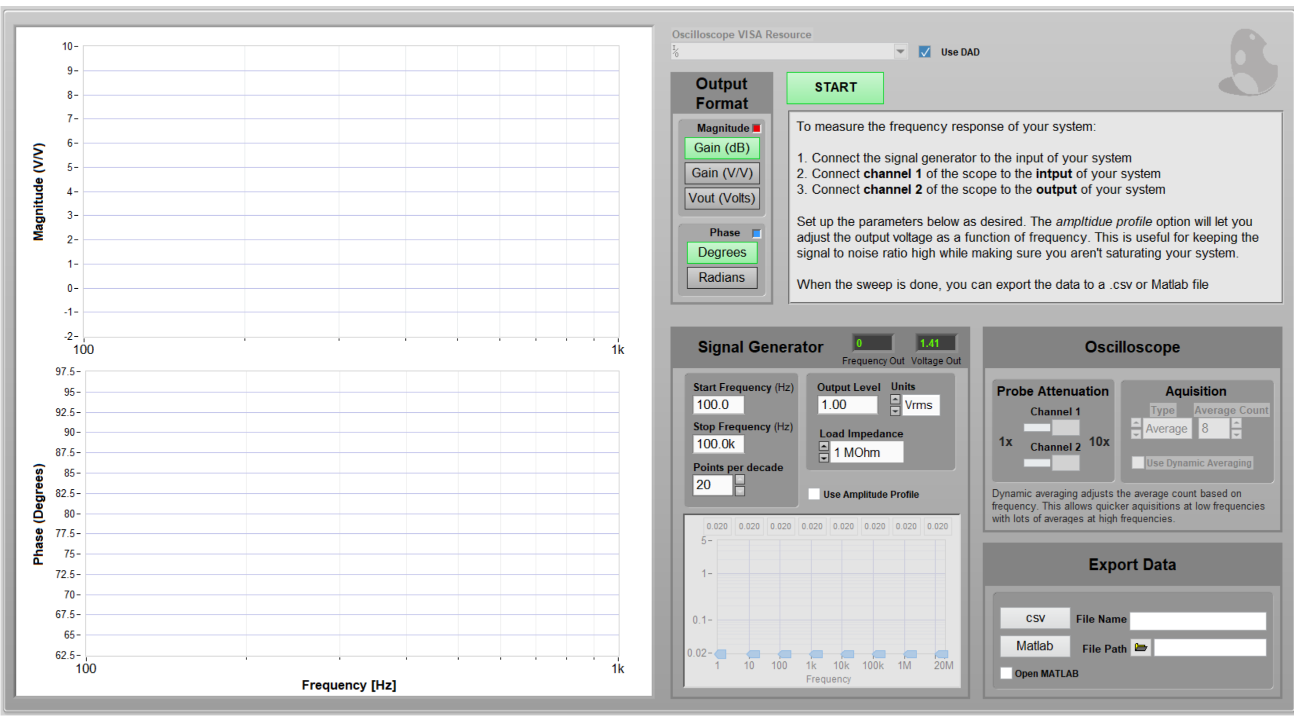


Figure 5.4: Loopinator Interface

The `Loopinator Software` won't work with the DAD while the `Waveforms Software` is open, so make sure it is closed before starting. Check the **Use DAD** checkbox and set the **Signal Generator** start and stop frequencies to 100 Hz and 100 kHz respectively. Enter 60 for **Points per decade**. Leave the output level at 1 V. Click **Start** and you should see the frequency response being plotted on the left. Once it has finished, choose a location to save the data; check the **Open MATLAB** checkbox, and click the **MATLAB Button**. Fig. 5.5 shows an example setup. MATLAB should open a file containing the swept data.

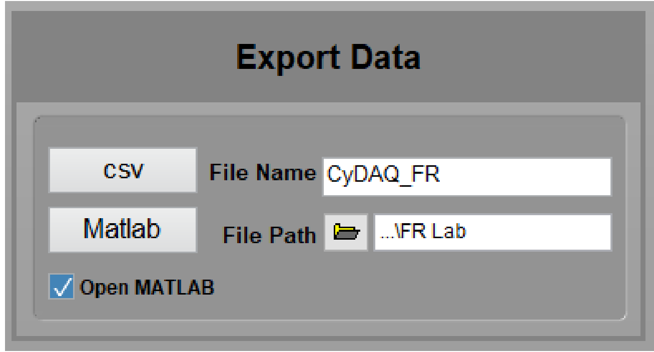


Figure 5.5: Loopinator Export Setup

5.4 Comparing Results in MATLAB

The MATLAB script generated by Loopinator is quite long with many lengthy variable definitions. We want to work with this data so we will be modifying this script. The first modification should be to add:

```
clearvars; close all; clc;
```

at the top of the Loopinator script. This will alleviate errors that crop up between successive runs of the script.

Navigate to the bottom of the script (below all of the variable definitions). Here, add code that will produce a figure consisting of two subplots: one depicting the magnitude in decibels and the other depicting the phase in degrees. Both of these should be plotted with respect to frequency on a semilog axis—the MATLAB *plot-like* function, `semilogx(...)` will be useful for this. All of the variables you need (the magnitude, phase, and frequency vectors) have been generated by Loopinator; you simply need to locate them in your script. Do not replace any code generated by Loopinator. Once you've added the code to produce this figure, run your script to confirm that it behaves as expected. Don't forget to title and label the axis of your plots.

Now, we will compare our manual measurements of the frequency response to the response computed by Loopinator. After the Loopinator variables but before the plotting code, create three vectors containing each of the columns of your Table 5.1. Since the Loopinator magnitude values we plotted are decibels, you will have to convert your gain measurements to decibels (refer

to Eq. (5.1)). Use your knowledge of MATLAB to compute the decibel equivalent of your gain measurements and store them in a *new* variable.

Modify the plotting section of your script to overlay scatter plots of your manual magnitude and phase measurements atop the respective Loopinator subplots. You will need to turn `hold on` for both subplots so that the `scatter(...)` function doesn't overwrite the Loopinator data. Once you've updated your code, run your script to confirm that it works as you would expect.

You may notice that your manual phase measurements seem inverted. This occurs if you had measured the time delays in Sec. 5.3.1 as positive numbers. If this was the case, it is acceptable to invert your manual phase measurements in the MATLAB code to better match the Loopinator values.

Include the final figure of your Loopinator magnitude and phase plots overlayed with your manual measurements in your lab report. Comment on the results. How close were your manual measurements to the values computed by Loopinator? What type of filter was deployed on the CyDAQ?

5.5 Report Checklist

Work with lab partner(s) to create a lab report for this lab exercise. Be sure to include the names of each person in your group on the report. This report should be a typed document that outlines the basic procedure of the lab exercise and should include any results you were asked to obtain. Your report should follow a *cohesive narrative*—it should always be apparent *why* you have presented the results you have. Do not simply insert results without (1) motivating the problem and (2) providing an analysis of the data. Ensure that your report is grammatically correct with complete sentences. This does not mean your report must be wordy with filler—in fact, avoid filler and state your ideas concisely.

Be sure the following are included in your report:

- Section 5.3.1: Explain the Δt to ϕ° formula of Eq. (5.3).
- Section 5.3.2: Fill out all the rows of Table 5.1 and include the complete table in your report. Discuss these results.
- Section 5.3.2: Based on your manual measurements, what type of filter do you believe has

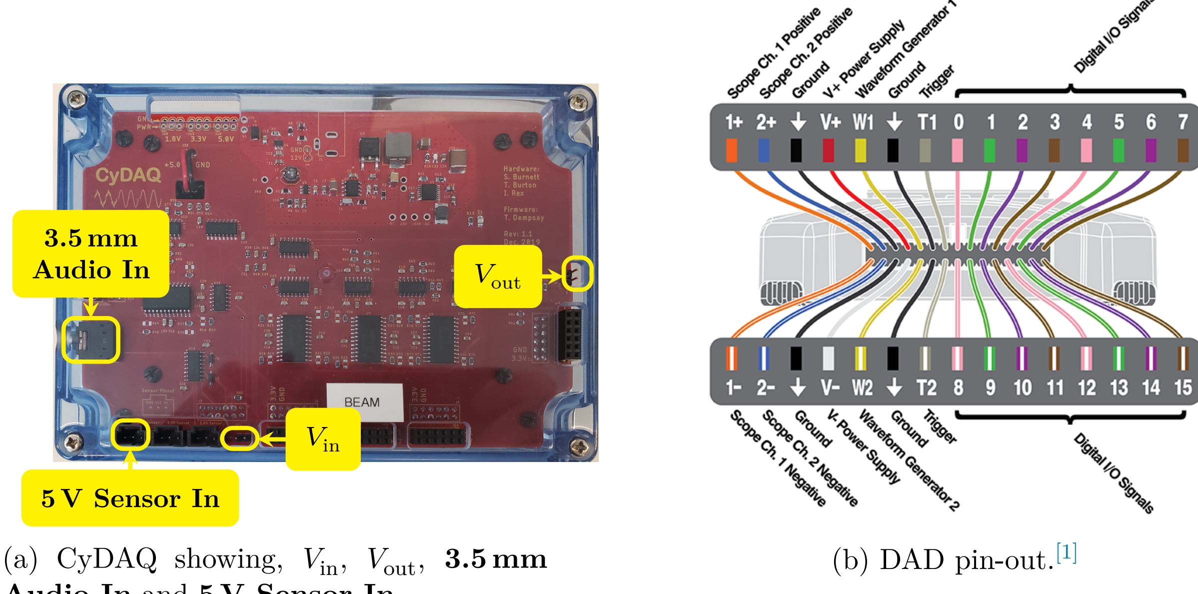
been deployed on the CyDAQ.

- Section 5.4: Include the final figure of your Loopinator magnitude and phase plots over-layed with your manual measurements in your report.
- Section 5.4: Discuss the accuracy of your manual measurements compared to the results calculated by Loopinator. What type of filter was deployed on the CyDAQ?

5.6 Acknowledgments

This lab exercise is based on an exercise drafted by **Isaac Rex**.

5.A Hardware Ports



(a) CyDAQ showing, V_{in}, V_{out}, **3.5 mm Audio In** and **5 V Sensor In**.

(b) DAD pin-out.[1]

Figure 5.6: Pin-outs/terminals for CyDAQ and DAD.

5.R References

[1] "Analog discovery 2 reference manual," Digilent. (2018), [Online]. Available: http://digilent.com/reference/test-and-measurement/analog-discovery-2/reference-manual.

Fourier Series Part I

The Math Behind the Music

Overview

In this lab exercise, students will use a collection of MATLAB scripts (provided in the *supplementary materials* for this exercise or Appendices 6.A–6.E) to compute the Fourier Series coefficients of periodic signals and synthesize signals given a set of Fourier Series coefficients. In addition to a typed report, students are asked to submit two generated audio files `piano_output.wav` and `scale_output.wav`.

Learning Objectives

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

1. Concatenate vectors and matrices of compatible sizes in MATLAB.
2. Compute the Fourier Series coefficients of a periodic signal in MATLAB.
3. Synthesize in a signal MATLAB given its Fourier Series coefficients.
4. Compute the negative Fourier Series coefficients of a real function given the positive coefficients.

6.1 Introduction

In this lab exercise, you will learn how to approximate Fourier Series coefficients using MATLAB. This will allow you to find the Fourier Series for any periodic signal, even if it does not have a form that can be represented in an equation.

6.2 MATLAB Background

6.2.1 Matrix Concatenation

In MATLAB it is possible to *concatenate* two or more *matrices* whose sizes are compatible. For instance, consider the matrices:

```
A = ones(2, 3);
B = zeros(2, 1);
C = 2*ones(2, 2);
```

It is possible to create a new matrix D consisting of the matrix, A, *horizontally concatenated* with the matrices B and C using the following command:

```
D = [A, B, C]
```

MATLAB produces the following output:

```
D =


     1     1     1     0     2     2
     1     1     1     0     2     2
```

This horizontal concatenation is possible because the matrices A, B, and C have the same number of rows. Attempting to horizontally concatenate matrices without the same number of rows:

```
A = ones(2, 3);
B = zeros(1, 3);
C = [A, B]
```

produces an error:

```
Error using horzcat
Dimensions of arrays being concatenated are not consistent.
```

It is also possible to *vertically concatenate* two or more matrices, permitted that each matrix has the same number of columns:

```
A = ones(2, 3);
```

```
B = zeros(1, 3);
C = [A; B]
```

MATLAB produces the following output:

```
C =

     1     1     1
     1     1     1
     0     0     0
```

6.2.2 Combining Concatenation with Array Slicing

Vector concatenation can be combined with *array slicing* to concisely generate new matrices from existing ones:

```
a = 1:3
b = [a(3:-1:1), 0, a]
```

MATLAB produces the following output:

```
b =

     3     2     1     0     1     2     3
```

6.2.3 Successive Concatenation

It is also possible to successively concatenate a matrix in order to add additional content. For example, we could add additional entries to a vector `x` as such:

```
x = [1, 1, 2, 3, 5];
x = [x, 8];
x = [x, 13, 21]
```

MATLAB produces the following output:

```
x =

```

```
3       1     1     2     3     5     8    13    21
```

This technique is incredibly inefficient, *however*, it is often useful for simple tasks and rapid prototyping.

6.3 Fourier Series and Musical Instruments

Music is just another signal. A musical note could represented by the function, $x(t)$, and may be any periodic signal. For a periodic musical note, $x(t)$, it is possible to approximate $x(t)$ using a sum of sinusoids.

6.3.1 Fourier Series Definition

Recall that the Fourier Series represents continuous-time, periodic signals as a sum of sinusoids:

$$x(t) = \sum_{k=-\infty}^{\infty} X_k \, e^{jk\omega_0 t}, \tag{6.1}$$

$$X_k = \frac{1}{T_0} \int_0^{T_0} x\left(t\right) e^{-jk\omega_0 t} \; \mathrm{d}t, \tag{6.2}$$

where $\omega_0 = 2\pi/T_0$.

From Eq. (6.1), we see that sinusoids are summed for k from $-\infty$ to ∞. *That is a lot of summing to do*—but most signals can be approximated by only a handful of X_k coefficients and sinusoids. For instance, a signal might be approximated using X_k for k from -5 to 5.

6.3.2 Fourier Series of a Triangle Wave

Let $x(t)$ be a $T_0 = 2$ periodic triangle wave signal given by:

$$x\left(t\right) = \begin{cases} t & 0 \leq t < 1 \\ 2 - t & 1 \leq t < 2 \\ \text{Repeat with a period of 2} & \forall t \end{cases} \tag{6.3}$$

Such a signal is depicted in Fig. 6.1.

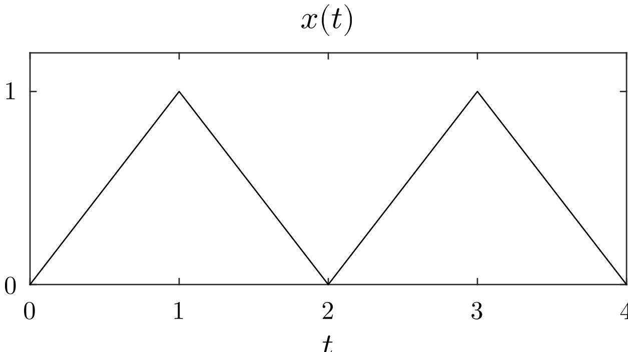


Figure 6.1: Triangle wave with a height of 1 and a period of $T_0 = 2$.

Manually compute the Fourier Series coefficients X_{-5} through X_5 for the triangle wave $x(t)$ using the Fourier Series analysis definition from Eq. (6.2) and include both your work and the solutions in your lab report. It is recommended that one partner work on computing these coefficients while the other continues forward in the lab exercise. If you are unable to compute these coefficients in *thirty minutes* or less—skip this prelab exercise and return to it once you have completed the remainder of the lab exercise.

Here are some tips for computing X_{-5} through X_5:

- The coefficient X_0 must be calculated independent from the others; in other words, you will need to compute:

$$X_0 = \frac{1}{T_0} \int_0^{T_0} x(t)\,\mathrm{d}t.$$

- Keep meticulous track of your $+/-$ signs; don't try to simplify too much in one step.
- You will need to use *integration by parts for definite integrals*; its formula is given as:

$$\int_a^b u\,\mathrm{d}v = \Big[\,u\,v\,\Big]_a^b - \int_a^b v\,\mathrm{d}u.$$

  When selecting expressions for u and $\mathrm{d}v$, you should choose u to be an expression that tends toward zero under repeated differentiation; for example, $u = e^t$ would be a ***poor*** choice since repeated differentiation of u does not tend toward zero.

The notation $[u\,v]_a^b$ is similar to the syntax for evaluating definite integrals once you have computed an anti-derivative; for instance, if u and v are functions of t, $u(t)$ and $v(t)$, then:

$$\Big[u(t)\,v(t)\Big]_a^b \;=\; u(b)\,v(b) - u(a)\,v(a).$$

- $\omega_0 = 2\pi/T_0$.
- After you have evaluated the integrals, you can use the relation $e^{-j\pi k} = (-1)^k$ to simplify your expressions.

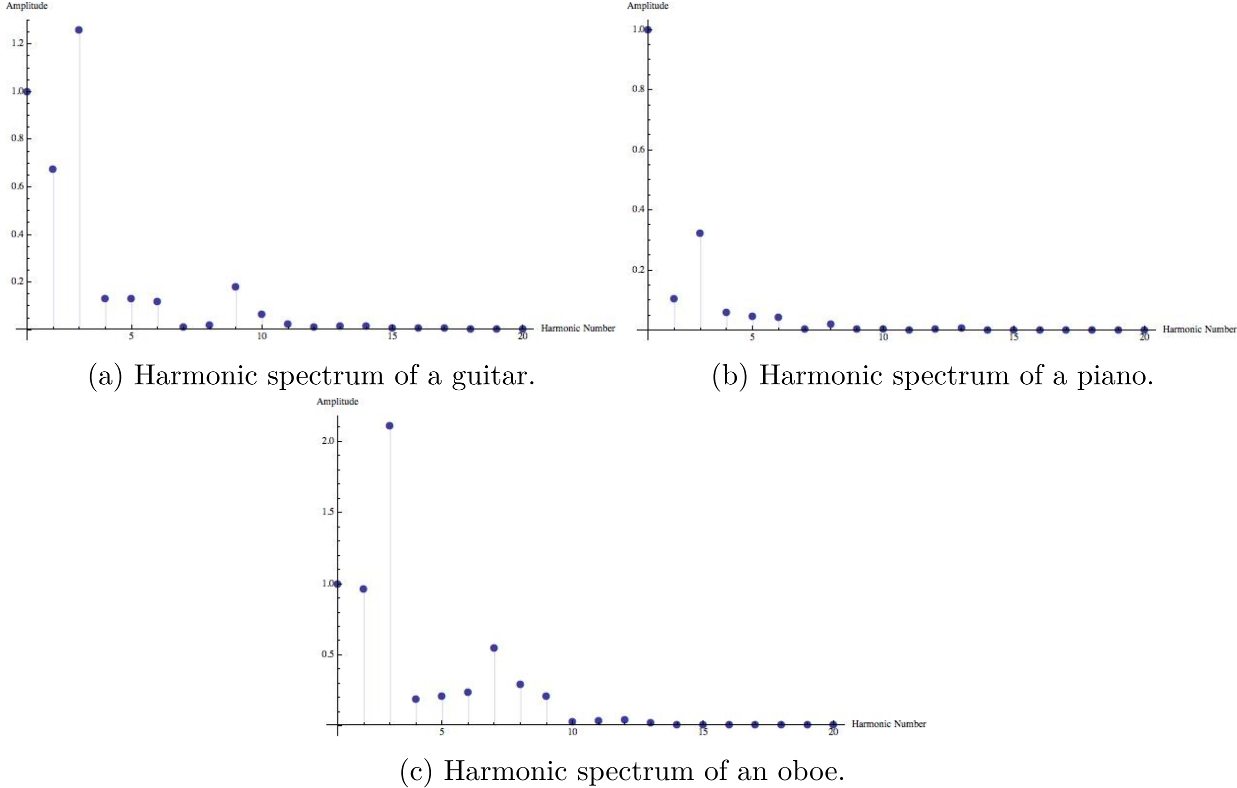
6.3.3 Relationship to Musical Instruments

Ever wonder why two musical instruments playing the same note can sound so drastically different? Each musical note corresponds to a frequency. If two instruments play the same note, their sound waves oscillate at the same fundamental frequency. They sound so different due to the varying harmonics between the two instruments. The difference in sound is called the "timbre" (pronounced *tam-ber*) of an instrument.

The first harmonic, or fundamental frequency, corresponds to the note being played. This harmonic must be easily heard when trying to match pitch or tune an instrument.

Musicians often listen to another instrument to tune their own. Since the fundamental frequency is the most important to listen to when tuning an instrument, which of the instruments from Fig. 6.2 should a musician tune to? Include this instrument and your reasoning in your lab report. *Hint*: Look at the value of each harmonic number in the charts below. The amplitude is the value of X_k where k is the harmonic number.

If the fundamental frequency that an instrument is playing is 440 Hz, then the next harmonics are at $k \times 440\,\text{Hz}$. Given a fundamental frequency of 440 Hz, compute the frequencies of the *second*, *third* and *fourth* harmonics and include them in your report.



(a) Harmonic spectrum of a guitar.

(b) Harmonic spectrum of a piano.

(c) Harmonic spectrum of an oboe.

Figure 6.2: Harmonic spectra of a guitar, piano and oboe; reproduced from Bell.[1]

6.3.4 Riemann Approximation of Fourier Series Coefficients

An integral is used when finding the values of X_k by hand. However, musical notes are nearly impossible to integrate. So, we will use a method of approximation called a Riemann Sum.

$$X_k \approx \frac{T_s}{T_0} \sum_{n=0}^{T_0/T_s} x\left(T_s n\right) \, e^{-jk\omega_0 T_s n}. \tag{6.4}$$

A Riemann Sum approximates the integral by breaking up the area under the graph into many small rectangles. Then, all the areas of the rectangles are added together. This will give a good enough approximation of the integral for our purposes. See Fig. 6.3 for reference.

Compare Fig. 6.3 with Eq. (6.4), where T_0 is the fundamental period of the signal, and T_s is the sampling period which corresponds to the inverse of the sampling frequency ($T_s = 1/f_s$). The function $x(t)$ is sampled into a discrete function $x[n]$ where each n corresponds to a sample number. Since $x[n]$ is an array of samples, T_s is the time between samples. Therefore $x(T_s n) =$

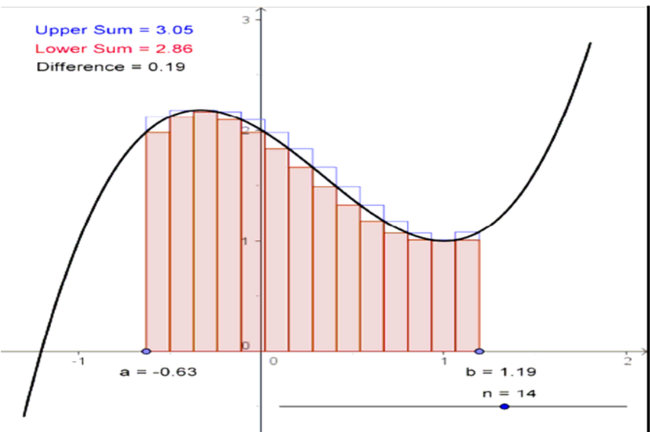


Figure 6.3: Riemann sum approximation of an integral; reproduced from Kllogjeri.[2]

$x[n]$. In other words, Eq. (6.4) takes the value of $T_s \cdot x[n] \cdot e^{-jk\omega_0 T_s n}$ to find the area of each rectangle in a Riemann Sum. After the area of each rectangle is found, all the areas are summed together.

Once the values for X_k have been found, an approximated signal can be reconstructed. The *truncated* Fourier Series synthesis definition:

$$\tilde{x}\,[n] = \sum_{k=-K}^{K} X_k\, e^{jk\omega_0\, n} \tag{6.5}$$

shows how a signal can be reconstructed. Notice that this equation is very similar to Eq. (6.1) except for the summation limits. The truncated synthesis definition, Eq. (6.5), recreates (or *synthesizes*) $x[n]$ by summing a set of sinusoids represented by $X_k \cdot e^{jk\omega_0\, n}$ between $-K$ and K. Consequentially, K is called the *order* of the re-synthesis.

6.4 Computing Fourier Series Coefficients using MATLAB

The following scripts are available in this exercise's *supplementary materials* (or Appendices 6.A–6.E): `find_Xk.m`, `find_Xks.m`, `fsynth.m`, `gen_square.m`, and `gen_tri.m`. These scripts are auxiliary MATLAB functions and cannot be run as standalone scripts. You can *call* and use the `help` command on them as you would any other function in MATLAB—provided they are visible

within your **Current Folder View**.

Create a new folder that will serve as your project directory and place each of the supplementary MATLAB functions into it. When this lab manual instructs you to create a new script, do so in this folder and ensure that this directory is the same directory visible in MATLAB's **Address Bar** and **Current Folder View**.

6.4.1 Approximating the Fourier Series of a Triangle Wave

We will now return to the triangle wave presented in Sec. 6.3.2 and tackle the problem of computing its Fourier Series coefficients, X_k, for $-5 \leq k \leq 5$ by employing MATLAB. This section will help familiarize you with many of the MATLAB functions included in the *supplementary materials* and walk you through what they do and how they work.

First, create a new standalone MATLAB script, `fs_triangle_wave.m`. Our first step is to recreate the triangle function given by Eq. (6.3). This describes a triangle wave that rises from 0 to a height of 1 and falls back to 0 over a period of $T_0 = 2$. The `gen_tri.m` MATLAB script can generate such triangle waves.

The `gen_tri(t, T0)` function takes two parameters: a time vector, `t`, and a scalar representing the period of the triangle wave, `T0`. Thus, we are tasked with determining the proper values of `t` and `T0` to generate a triangle wave that will be used to approximate Fourier Series coefficients. Determining the value of `T0` is straightforward; it is simply the period of the triangle wave, which is given as $T_0 = 2$.

Generating an appropriate time vector, `t`, requires a bit more thought. To accurately approximate Fourier Series coefficients, the triangle wave we generate should:

1. span exactly one period, and
2. consist of a finely sampled set of points such that the Riemann Sum from Eq. (6.4) produces an accurate approximation of the integral from Eq. (6.2).

This means that our time vector, `t`, should span from 0 to 2 (the length of one full period of the triangle wave) and have a very small distance between samples, or *sampling period*: T_s. A small sampling period implies that we have a large *sampling frequency*, f_s (as these two quantities are reciprocals). We will select $f_s = 22050$ as a sufficiently large value.

In your `fs_triangle_wave.m`, enter the following code:

Listing 6.1: `fs_triangle_wave.m`

```
clc; close all; clearvars;

T0 = 2; % the period of the triangle
T_dur = 2; % the duration of the triangle (we only want a single period)
Fs = 22050; % the sampling rate of the triangle wave
Ts = 1/Fs;

% Generate the triangle wave
t = 0:Ts:T_dur; % time vector of duration T_dur with sample period of T_s
x_tri = gen_tri(t, T0); % call the provided gen_tri script

% Plot the triangle wave
figure(1);
plot(t, x_tri)
xlabel('t')
title('Triangle Wave x(t)')
```

Run the script. MATLAB should produce a plot of a single period of the triangle wave. Review the script you've written so far and ensure you understand everything. In particular, ensure that you understand how the time vector, `t`, was derived and what the values T_s, T_0 and T_{dur} represent. An illustration of how each of these values relate to one another is given in Fig. 6.4.

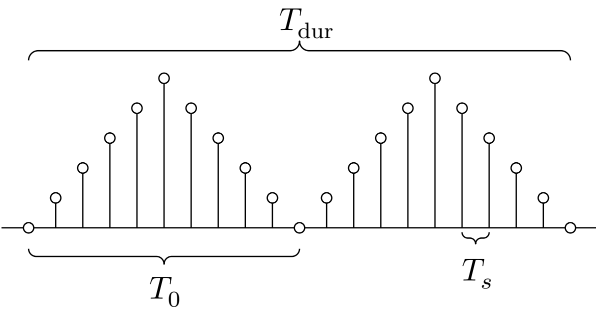


Figure 6.4: An illustration of what the parameters T_s, T_0 and T_{dur} from Listing 6.1 represent.

Note that the illustration given in Fig. 6.4 *will not* match the plot generated by the MATLAB script. For one, Fig. 6.4 depicts two periods of the triangle wave while the MATLAB script only generates one period. Secondly, the illustration depicts a value of T_s which is much too large for the Riemann Sum of Eq. (6.4) to offer an acceptable approximation of Eq. (6.2).

Now we will compute the Fourier Series coefficients, X_k, for $-5 \leq k \leq 5$. This can be accomplished using the `find_Xks.m` MATLAB file. The `find_Xks(x, K, Ts)` function takes three parameters: a vector, `x`, containing a single period of a periodic signal $x(t)$ that we wish to compute the Fourier Series coefficients of; a scalar, `K`, representing the largest/smallest Fourier Series coefficient to compute; and a scalar, `Ts`, specifying the *sampling period* of `x`. The `find_Xks()` function returns a vector, $\mathbf{X}_{ks}$, consisting of the Fourier Series coefficients, X_k, for $-K \leq k \leq K$:

$$\mathbf{X}_{ks} = \begin{bmatrix} X_{-K} & \cdots & X_0 & \cdots & X_K \end{bmatrix}. \tag{6.6}$$

Add the following code to your `fs_triangle_wave.m` script:

```
19 % Find the Fourier Series coefficients, X_ks, of the triangle wave
20 K = 5; % The largest/smallest Fourier Series coefficients to compute
21 Xks = find_Xks(x_tri, K, Ts);
```

Run the MATLAB script. This should approximate the Fourier Series coefficients, X_k, for $-5 \leq k \leq 5$ and store them in the vector `Xks`.

Now let's compare the approximated coefficients' values to those you computed manually in Sec. 6.3.2. To aid us in this effort, let's plot the real portion of each coefficient approximated by MATLAB (if everything was done properly, these coefficients should be "mostly real" anyways, with any imaginary portion being a product of numerical error or imprecise approximation). Add the following code to your `fs_triangle_wave.m` script:

```
23 k = -K:K;
24 figure(2)
25 stem(k, real(Xks))
26 title('Re(X_k)')
27 xlabel('k')
```

Run the script. Compare the results of MATLAB's approximation to the values you computed manually. Comment on this in your lab report.

Finally, we will re-synthesize a Fourier Series approximation of the original triangle wave using our computed Fourier Series coefficients. This can be accomplished with the `fsynth.m` MATLAB file.

The `fsynth(Xks, T0, Ts, T_dur)` function takes four parameters: a vector, `Xks`, consisting of the Fourier Series coefficients X_k; a scalar, `T0`, containing the fundamental period (reciprocal to the fundamental frequency f_0) we wish to synthesize; a scalar, `Ts`, containing the sampling frequency of the synthesized signal; and a scalar, `T_dur`, containing the total duration of the sample we wish to synthesize. The `fsynth` function returns a vector, $\tilde{x}(t)$, containing the Fourier series approximation of $x(t)$.

The ability to choose T_0 allows us to synthesize signals with fundamental frequencies different from that of the original signal—this comes in handy for say, synthesizing instruments at different note pitches—but in our case, we will simply re-synthesize the signal using the original period, $T_0 = 2$.

The ability to choose T_s allows us to synthesize signals with sampling periods that differ from the original signal. In our case, we will continue to use a sampling period of $T_s = 1/22050$.

The ability to choose T_{dur} lets us synthesize signals with total durations that differ from the original signal. For the sake of example, let's choose $T_{\text{dur}} = 4$, equivalent to 2 periods.

Add the following code to your `fs_triangle_wave.m` script:

```
30 % Re-synthesize the tri wave
31 T_dur = 4; % now with a duration of 4 sec
32 x_hat = fsynth(Xks, T0, Ts, T_dur);
33 t_hat = 0:Ts:T_dur;
34
35 figure(3)
36 plot(t_hat, x_hat)
37 xlabel('t')
38 title('Re-synthesized x(t)')
```

Run the script. MATLAB should produce a plot of the Fourier Series approximation of 2 periods

of a triangle wave. Does it look how you would expect?

6.4.2 Re-synthesizing a Square Wave

Using the `fs_triangle_wave.m` script as your guide, create a new script, `fs_square_wave.m` which uses the `gen_square`, `find_Xks` and `fsynth` functions to generate Fourier Series approximations of square waves for $K = 2$, 10, and 100. Plot the signals over three periods on the same figure using separate subplots. Comment on the results. Use the default sampling period $T_s = 1/22050$ and default fundamental period $T_0 = 1$ where needed. When generating a signal, `x`, to pass to the function `find_Xks(...)`, ensure that `x` spans only one period.

Once you have re-synthesized the square waves for $K = 2$, 10, and 100, you'll notice that the synthesized signal overshoots the amplitude value 1 and undershoots the amplitude value 0. Measure the peak values of the square wave overshoot when the signal transitions from 1 to 0 and 0 to 1 for all three cases. Make a table of the values. Describe what you observe. Does this overshoot/undershoot get better or worse for increasing values of K?

6.4.3 Re-synthesizing a Piano

In this section, you will create a simplified version of a musical note for a piano. In music, timbre is the perceived sound quality of a musical note, sound, or tone. Timbre distinguishes different types of sound production, such as choir voices and musical instruments (strings, wind, and percussion). The physical characteristics of sound that determine the perception of timbre include spectrum and envelope (describes how a sound changes over time). In this lab exercise, we focus on the spectral harmonics and the tonal character of the underlying periodic sound.

Suppose we are given the following Fourier Series coefficients for a piano: $X_0 = 0$, $X_1 = 0.1$, $X_2 = 0.33$, $X_3 = 0.06$, $X_4 = 0.05$, $X_5 = 0.045$, $X_6 = 0$, $X_7 = 0.02$, $X_8 = 0.005$, $X_9 = 0.005$, $X_{10} = 0$, $X_{11} = 0.005$, $X_{12} = 0.01$. Suppose that we are asked to use these coefficients to synthesize a piano note which sounds A_4 (440 Hz) for 2 seconds, with a sampling frequency of $f_s = 22050$.

From the previous sections, we know that we can use the `fsynth(...)` function to generate such a signal. We know that to generate a note with a fundamental frequency $f_0 = 440$, we

would need to select a fundamental period of $T_0 = 1/440$. We are also given T_s and T_{dur}. There is still one big problem: we are not given the complete Fourier Series coefficient vector, $\mathbf{X}_{ks}$. Rather, we are only given the coefficients at positive indices: X_0 through X_{12}. We are still missing the negative indices: X_{-12} through X_{-1}.

Luckily, there is a property we can exploit which will give us the values of these coefficients at negative indices. We know (or rather, we assume) that the time-domain piano signal is real. Real functions in the time domain are conjugate symmetric in the frequency domain.[3]:204 This means that if your $x(t)$ signal is real (which your piano sound is) and you know X_k for $k > 0$, you get the $k < 0$ terms *for free*—they are given by:

$$X_{-k} = \text{conj}(X_k) \qquad \text{for all} \quad k \neq 0,$$

where $\text{conj}(X_k)$ refers to taking the *complex conjugate*[4] of X_k.

Create a new script, `synthesize_piano.m`, and fill-in the following code:

Listing 6.2: `synthesize_piano.m`

```
clc; close all; clearvars;

F0 = 440;
T0 = 1/440;
T_dur = 2;
Fs = 22050;
Ts = 1/Fs;

X0 = 0;
X_ks_rh = [0.1, 0.33, 0.06, 0.05, 0.045, 0, 0.02, 0.005, 0.005, 0, 0.005, 0.01];
X_ks = [TODO]; % <- fill in the TODO, use vector concatenation and array slicing

x_hat = fsynth(TODO); % <- fill in TODO

% we need this so this sound data doesn't exceed +/-1 and clip
x_hat = x_hat./max(abs(x_hat));

plot(x_hat)
```

```
19
20  sound(x_hat, Fs);
21  audiowrite('piano_output.wav', x_hat, Fs)
```

Complete each `TODO` portion of the above script. *Hint*: You can use matrix concatenation and array slicing to define the vector `X_ks`.

Once you have completed the script, run it in MATLAB. Adjust the viewing region of the plot (using the `axis` command) so that exactly one period of the piano signal is visible. Include the plot of a single period of the piano signal in your lab report. Include the resulting `piano_output.wav` as an attached file in your submission.

Synthesizing a Piano Scale

Similar to the previous section, we will use the same Fourier Series coefficients to synthesize a *scale* of piano notes using the frequencies in Tab. 6.1 with each note lasting 1 second. Create a new script called `synthesize_piano_scale.m` and enter the following code:

Listing 6.3: `synthesize_piano_scale.m`

```
1   clc; close all; clearvars;
2
3   noteF0s = [523.25, 587.33, 659.25, 698.46, 783.99, 880.00, 987.77, 1046.50];
4
5   T_dur = 1; % duration of each note
6   Fs = 22050;
7   Ts = 1/Fs;
8
9   X0 = 0;
10  X_ks_rh = [0.1, 0.33, 0.06, 0.05, 0.045, 0, 0.02, 0.005, 0.005, 0, 0.005, 0.01];
11  X_ks = [TODO]; % <- fill in the TODO, use vector concatenation and array slicing
12
13  x_scale = [];
14  x_scale = [x_scale, fsynth(TODO)]; % <- fill in the TODO
15  % repeat 'x_scale = [x_scale, fsynth(TODO)];' 7 more times
16  % for a total of 8 notes in the C-major scale
17  % ...
```

```
18 % ...
19
20 % we need this so this sound data doesn't exceed +/-1 and clip
21 x_scale = x_scale./max(abs(x_scale));
22
23 figure(1);
24 plot(x_scale)
25 xlabel('t')
26 title('C-major scale on piano')
27
28 sound(x_scale, Fs); % play the sound
29 audiowrite('scale_output.wav', x_scale, Fs)
```

Note	Frequency (Hz)
C_5	523.25
D_5	587.33
E_5	659.25
F_5	698.46
G_5	783.99
A_5	880.00
B_5	987.77
C_6	1046.50

Table 6.1: Note frequencies of a C major scale; reproduced from Suits.[5]

Complete the `TODO` sections of the code. You will need to use *successive concatenation* with the vector `x_scale` to synthesize a complete scale. Include the resulting `scale_output.wav` as an attached file in your submission. Comment on what you hear.

6.5 Report Checklist

Work with lab partner(s) to create a lab report for this lab exercise. Be sure to include the names of each person in your group on the report. This report should be a typed document that outlines the basic procedure of the lab exercise and should include any results you were asked to obtain. Your report should follow a *cohesive narrative*—it should always be apparent *why* you

have presented the results you have. Do not simply insert results without (1) motivating the problem and (2) providing an analysis of the data. Ensure that your report is grammatically correct with complete sentences. This does not mean your report must be wordy with filler—in fact, avoid filler and state your ideas concisely.

Be sure the following are included in your report:

- Section 6.3.2: Using the Fourier series analysis definition in Eq. (6.2), manually compute the Fourier Series coefficients X_{-5} through X_5 for the triangle wave signal given by Eq. (6.3). Provide both the result and your work in the lab report.
- Section 6.3.3: Based on the spectra from Fig. 6.2, determine which instrument is the best for a musician to tune to. Provide your reasoning.
- Section 6.3.3: Given a fundamental frequency of 440 Hz, compute the frequencies of the *second*, *third* and *fourth* harmonics.
- Section 6.4.1: Approximate the coefficients of the triangle wave from Eq. (6.3) in MATLAB using the provided `gen_tri` and `find_Xks` functions. Compare this result to the coefficients you computed manually in the pre-lab.
- Section 6.4.1: Provide a stem-plot of the *real valued portion* of MATLAB coefficients, X_{-5} through X_5, you computed for the triangle wave from Eq. (6.3).
- Section 6.4.2: Re-synthesize an approximation of a square wave for $K = \{2, 10, 100\}$ and include a figure containing subplots of each re-synthesized signal over three periods. Comment on the results.
- Section 6.4.2: For each of your re-synthesized square waves ($K = \{2, 10, 100\}$), measure the peak values of the overshoots and undershoots and make a table of their values. Describe what you observe.
- Section 6.4.3: Re-synthesize a piano note at 440 Hz with a sampling rate of $f_s = 22050$ using the provided coefficients. Provide a plot of a single period of the synthesized piano note.
- Section 6.4.3: Include the re-synthesized piano note, `piano_output.wav`, as an attached file in your submission.
- Section 6.4.3: Include the synthesized notes of a C-major scale, `scale_output.wav`, as an attached file in your submission.

6.6 Acknowledgements

This lab exercise is based on an exercise drafted by **Taylor Burton** in co-operation with **Andrew Bolstad**.

6.A Generate Triangle Wave Function

Listing 6.4: gen_tri.m

```
function x = gen_tri(t, T0)
% GEN_TRI generates a T0 periodic balanced triangle wave.
%   x = GEN_TRI(t, T0) balanced triangle wave with a period of
%   T0 seconds with respect to time vector t.
%
% Usage example:
%   Ts = 1/22050;
%   t = 0:Ts:10;
%   x = gen_tri(t, T0);

% Written By: Julie Dickerson, 2020
% Updated By: Aaron Fonseca, 2024


    t_falling = T0 / 2;
    tri_slope = 1 / t_falling;

    tri_func = @(t) ...
        (t <= t_falling).*(t .* tri_slope) + ...
        (t > t_falling).*(1 - ((t - t_falling) .* tri_slope));

    x = tri_func(mod(t, T0));
end
```

6.B Generate Square Wave Function

Listing 6.5: `gen_square.m`

```
function x = gen_square(t, T0)
% GEN_SQUARE generates a T0 periodic 50 percent duty cycle square wave.
%   x = GEN_SQUARE(t, T0) generates a 50 percent duty cycle square wave
%   with a period of T0 seconds with respect to time vector t.
%
% Usage example:
%   Ts=.0001; T0=4; Tdur=8;
%   t = 0:Ts:Tdur;
%   x = gen_square(t, T0);

% Written By: Julie Dickerson, 2020
% Updated By: Aaron Fonseca, 2024


    square_func = @(t) 0.*t + 1.*(t<=T0/2);
    x = square_func(mod(t, T0));
end
```

6.C Compute Fourier Series Coefficients Function

This function requires the `find_Xk.m` function (see Appendix 6.D) to run.

Listing 6.6: `find_Xks.m`

```
function X_ks = find_Xks(x, largestK, Ts)
% FIND_XKS Computes the Fourier series coefficients of x between +/-largestK.
%   X_ks = FIND_XKS(x, largestK, Ts) computes the Fourier seires coefficients
%   between -largestK and largestK of the signal x, where largestK is a
%   positive integer, x is a column vector containing a *single* period of a
%   function and Ts is the sampling period of x.

```

```
% Written By: Julie Dickerson, 2020
% Updated By: Aaron Fonseca, 2024


    % Set up the coefficient index vector
    ks = -largestK:largestK;

    % setup vector for results
    numK = length(ks);
    X_ks = zeros(size(ks));

    % loop through all coefficients to compute coefficients
    for i = 1:numK
        X_ks(i) = find_Xk(x, ks(i), Ts);
    end
end
```

6.D Compute Fourier Series Coefficient Function

Listing 6.7: find_Xk.m

```
function X_k = find_Xk(x, k, Ts)
% FIND_XK Computes a single Fourier series coefficient via Riemann approx.
%   X_k = FIND_XK(x, k, Ts) computes the kth Fourier series coefficient of the
%   signal: x, where x is a column vector containing a *single* period of a
%   function and Ts is the sampling period of x.

% Written By: Julie Dickerson, 2020
% Updated By: Aaron Fonseca, 2024


    % Ensure that x is a column vector
    [rx, ] = size(x);
    if (rx==1)
```

```
14        x=x.';
15     end
16 
17     % Compute the length of the period, T.
18     T = length(x) * Ts;
19     w0 = (2*pi) / T;
20 
21     t = 0:length(x)-1;
22     t = t * Ts;
23 
24     % Compute the complex part over all values of t.
25     basis_func = exp(-1j*k*w0*t);
26 
27     % The complex part mulitiplied by the function yields the Riemann sum.
28     X_k = (Ts/T) * basis_func*x;
29 end
```

6.E Synthesize Fourier Series Function

Listing 6.8: fsynth.m

```
1  function x_hat = fsynth(X_ks, T0, Ts, Tdur)
2  % FSYNTH Synthesizes a time-domain signal given Fourier series coefficients.
3  %   x_hat = FSYNTH(X_ks, T0, Ts, Tdur) synthesizes 'Tdur' seconds of a T0
4  %   periodic signal, with a sampling period Ts, specified by the Fourier
5  %   series coefficients given in X_ks.
6  %
7  % Inputs
8  %   X_ks: a 2K+1 length vector of the form:
9  %            [X[-K] ... X[-1], X[0], X[1] ... X[K]]
10 %        where X[k] gives the kth Fourier series coefficient of the signal
11 %        to synthesize.
12 %   T0: a scalar specifying the fundamental period of the signal to synthesize.
13 %   Ts: a scalar specifying the sampling period of the signal to synthesize.
```

```matlab
14 %   Tdur: a scalar specifying the total duration (in seconds) of the signal to
15 %         synthesize.
16
17 % Written By: Julie Dickerson, 2020
18 % Updated By: Aaron Fonseca, 2024
19
20
21     if rem(numel(X_ks), 2)  = 1
22         error('numel(X_ks) = %d. The length of the X_ks vector should be odd.', numel(X_ks))
23     end
24     K = floor((numel(X_ks) - 1) / 2);
25     ks = -K:K;
26
27     % Find omega wich is (2*pi)/Period
28     % Frequency is the inverse of the Period
29     % T0 is the fundamental period of the function
30     w0 = (2*pi)/T0;
31
32     % make time vector for evaluation
33     t = 0:Ts:Tdur;
34
35     % Compute the Fourier approximation
36     x_hat = real(X_ks * exp(1j*w0*ks'*t));
37 end
```

6.R References

[1] M. Bell. "Fourier analysis in music," Project Rhea, Purdue University. (2023), [Online]. Available: http://www.projectrhea.org/rhea/index.php/Fourier_analysis_in_Music.

[2] P. Kllogjeri, "Geogebra: A global platform for teaching and learning math together and using the synergy of mathematicians," in *Technology Enhanced Learning. Quality of Teaching and Educational Reform*, M. D. Lytras, P. Ordonez De Pablos, D. Avison, *et al.*, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 681–687.

[3] F. Ulaby and A. Yagle, *Signals and Systems: Theory and Applications*. Michigan Publishing, 2018, ISBN: 978-1-60785-487-6.

[4] "Complex conjugate," MathWorks Inc. (2023), [Online]. Available: http://www.mathworks.com/help/matlab/ref/conj.html.

[5] B. H. Suits. "Physics of music notes, musical note frequencies for an equal tempered scale," Physics Department, Michigan Technological University. (2023), [Online]. Available: http://pages.mtu.edu/~suits/notefreqs.html.

Fourier Series Part II

The Math Behind the Music

Overview

In this lab exercise, students will use a collection of MATLAB scripts (provided in the *supplementary materials* for this exercise or Appendices 7.A–7.E) and a collection of recorded instrument samples (provided in this exercise's *supplementary materials*) to compute the Fourier Series coefficients of various instruments. Students will then use these Fourier Series coefficients to re-synthesize instruments playing notes at various pitches and durations. In addition to a typed report, students are asked to submit four audio files: `piano_resynth.wav`, two re-synthesized instruments of their choice, and `chrom_scale.wav`.

Learning Objectives

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

1. Repeat/tile matrices along one or more axes in MATLAB.
2. Locate locally periodic regions of instrument recordings.
3. Use MATLAB to compute the Fourier Series coefficients of an instrument recording.
4. Use MATLAB to re-synthesize a recorded instrument at various pitches/durations using its Fourier Series coefficients.

7.1 Introduction

In this lab exercise, you will cover how to estimate Fourier Series coefficients from instrument recordings. You will then create a basic music synthesizer.

7.2 MATLAB Background

In MATLAB it possible to *repeat* (or *tile*) a matrix along one or more axes using the `repmat(...)` function.[1] For instance, consider the matrix:

$$\mathbf{A} = \begin{bmatrix} -1 & 0 & 1 \\ 1 & 2 & 3 \end{bmatrix}, \tag{7.1}$$

which is defined in MATLAB using the following command:

```
A = [-1, 0, 1; 1, 2, 3]
```

Suppose we wanted to create a matrix, **B**, which consists of the matrix **A** repeated/tiled *three* times horizontally:

$$\mathbf{B} = \begin{bmatrix} \mathbf{A} & \mathbf{A} & \mathbf{A} \end{bmatrix} = \begin{bmatrix} -1 & 0 & 1 & -1 & 0 & 1 & -1 & 0 & 1 \\ 1 & 2 & 3 & 1 & 2 & 3 & 1 & 2 & 3 \end{bmatrix}. \tag{7.2}$$

One way to accomplish this would be to use *matrix concatenation*, but another way to accomplish this is to use the `repmat(...)` function.

The `repmat(A, m, n)` function takes three parameters: the matrix, `A`, containing the matrix to be repeated; the scalar, `m`, containing the number of times to repeat `A` *vertically*; and the scalar, `n`, containing the number of times to repeat `A` *horizontally*.

Initializing **B** from Eq. (7.2) using MATLAB can be done with the following command:

```
B = repmat(A, 1, 3)
```

MATLAB produces the following output:

```
B =

   -1     0     1    -1     0     1    -1     0     1
    1     2     3     1     2     3     1     2     3
```

Suppose we wanted to initialize a new matrix, **C**, which consists of the matrix **A** repeated *one hundred* times *vertically*. Give the MATLAB command (leveraging `repmat(...)`) to initialize

such a matrix in your report.

7.3 Background

In the previous lab exercise, we learned about timbre (pronounced *tam-ber*) and how the timbre of various instruments is defined by the harmonics present when an instrument sounds a note. We learned that *magnitude* and *phase* of these harmonics are encoded within the Fourier Series (FS) coefficients X_k. We then derived the FS coefficients for square and triangle waves and re-synthesized them using a *truncated* Fourier Series. Finally, we were given the FS coefficients X_k for $k \geq 0$ for a piano, which we used to synthesize a C-Major scale.

In this lab exercise, we will focus more on re-synthesizing instruments using truncated Fourier Series and how to go about obtaining FS coefficients from recordings of instruments. The *supplementary materials* for this exercise provide a large collection of instrument recordings. All of the recordings *expect* `piano.wav` sourced from the *Music Technology Group* (MTG) of the Universitat Pompeu Fabra in Barcelona as part of the good-sounds.org project.[2] The `piano.wav` recording was sourced from TEDAGame's 88 Piano Keys Pack.[3] The *supplementary materials* for this exercise also provide the following MATLAB files: `cut_sample.m`, `find_Xk.m`, `find_Xks.m`, `fsynth.m`, and `plot_pitch.m`. These files are also available in Appendices 7.A–7.E. You should be familiar with the functions provided by: `find_Xk.m`, `find_Xks.m`, and `fsynth.m` from the Fourier Series Part I exercises—please refer back to those for reference (if needed). The `cut_sample.m` and `plot_pitch.m` MATLAB files are new to this lab exercise. Note that the `plot_pitch.m` script utilizes the `pitch(...)` function, which is only available if you have the Audio Toolbox (v2019) add-on[4] installed.

All the MATLAB files included provide functions and cannot be executed as stand-alone scripts (i.e., you cannot execute them by pressing the **Run** button). Instead, these MATLAB files provide functions that can be called so long as they are present in MATLAB's working directory (i.e., visible in your **Current Folder View**).

7.4 Re-synthesizing Instruments

We have been provided with a collection of different instrument recordings. This exercise will outline the process of creating synthesizable instrument *patches* for a subset of these instrument recordings. These patches are sets of truncated Fourier Series, X_k, which encapsulate the timbre of individual instruments. This is similar to the process of computing the FS coefficients for the triangle and square waves of the previous exercise, except this time, rather than basic shapes, we will be computing the FS for instruments.

7.4.1 Isolating Slices from Recordings

The first step in this process is to isolate a *slice* of the instrument waveform. A *slice* is a cut of the instrumental waveform containing a single 'period' of a localized region of the waveform. The reason *period* is in quotes is because instrumental recordings are not strictly periodic. When instruments sound a given note, the total waveform can usually be partitioned into distinct regions: an *attack* region, where the instrument swells from silence to its loudest point; a *sustain* region, where the instrument maintains a relatively constant volume; and a *decay* region, were the instrument fades to silence.

Within the *sustain* region of an instrument, the waveform data is locally periodic (or close to periodic). The period in this region, T_0, will be reciprocal to the fundamental frequency, f_0, of the note the instrument is sounding. To locate a slice, we must first locate the *sustain* region of an instrument recording and then isolate a single period from that region.

This is best explained through example. Create a new folder that will serve as your project directory and place each of the supplementary MATLAB functions into it. Locate `piano.wav` in the *supplementary materials* and place it into your project directory. Launch MATLAB in your project folder and make sure that all of the supplementary scripts and `piano.wav` are visible within the **Current Folder** view (see Fig. 7.1).

Create a new script in your project folder called `piano_slice.m`. We will use this script to explore the process of isolating a slice from `piano.wav`. To do so, we will leverage the included `plot_pitch(...)` function. The `plot_pitch(x, Fs, plotTitle)` function takes three parameters:

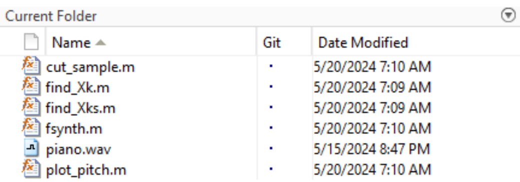


Figure 7.1: The current folder view showing the included scripts and `piano.wav`.

the matrix, `x`, containing the waveform data of the instrument recording; the scalar, `Fs`, containing the sampling frequency of the instrument recording; and the character array, `plotTitle`, containing a title for the figure this function produces. Once called, this function will populate a figure with two subplots: the first depicting the waveform data of the instrument recording with respect to the current sample, n, (e.g., it depicts the audio waveform $x[n]$); and the second subplot depicts the estimated fundamental frequency, f_0, of the waveform with respect to the current sample, n. Both subplots share the same x-axis.

In your `piano_slice.m` script, enter the following code:

Listing 7.1: `piano_slice.m`

```
clc; clearvars; close all;

wav_filepath = 'piano.wav';
[x, Fs] = audioread(wav_filepath);
x = x(:,1); % grab only the first channel

figure(1);
plot_pitch(x, Fs, 'piano.wav')
```

Run the MATLAB script. You should see a result similar to that depicted in Fig. 7.2. Note that in these subplots, the x-axis is measured by sample number (n), not time (t).

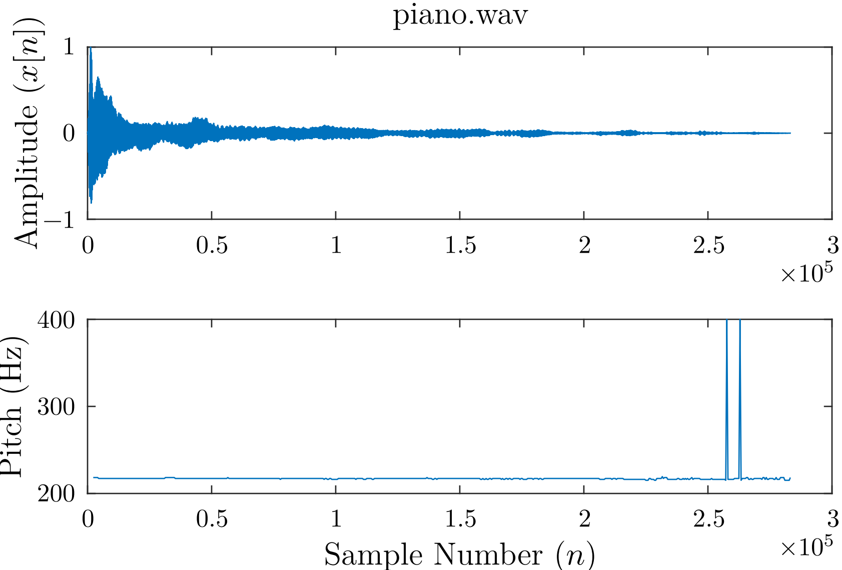


Figure 7.2: Result of the `plot_pitch(...)` function for `piano.wav`; the upper subplot depicts the waveform data of the piano signal while the lower subplot depicts the estimated pitch of the fundamental frequency at each sample; both plots share the same x-axis.

Our goal now is to locate the *sustain* region of the waveform. This can be done using both subplots produced by `plot_pitch(...)`. We want to locate a region where:

1. the amplitude (upper subplot) envelope is mostly constant (flat), and
2. the estimated pitch (lower subplot) is mostly constant.

For the piano recording in Fig. 7.2, the estimated pitch remains mostly constant around 217 Hz for $n < 2.5 \times 10^5$ but there isn't an easily identifiable region of the amplitude waveform that remains around the same volume (i.e., having a flat amplitude envelope).

Upon closer inspection, the amplitude envelope appears mostly constant around $n = 5350$. This isn't directly apparent in Fig. 7.2, but can be seen by zooming in on the graph (I determined the n value by clicking on points on the graph and examining the x-axis). Consequentially, the region around $n = 5350$ has a high likelihood of containing our desired slice.

As mentioned earlier, a slice is a small cut of an instrument waveform that is roughly characteristic of localized periodicity. As such, we can define a slice using two parameters: the sample number at which the slice begins, n_a, and the total number of samples within a slice, N. An illustration of these parameters as they relate to waveform data is given in Fig. 7.3. Once we've determined these two parameters, we can use the `cut_sample(...)` function to retrieve

the waveform data of our slice. The `cut_sample(x, n_a, N)` function takes three parameters: a matrix, `x`, containing the waveform data we wish to slice; a scalar `n_a`, containing the starting sample of the slice; and a scalar, `N`, containing the total number of samples in the slice.

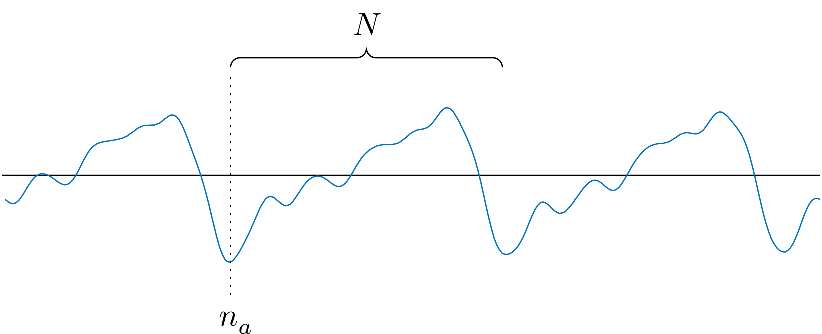


Figure 7.3: Illustration of the parameters: n_a, representing the starting sample; and N, representation the total number of samples within a single period of the fundamental frequency, f_0.

Since we've identified the region around $n = 5350$ as a likely candidate for where our slice is located, we will choose $n_a = 5350$. To obtain N, we will exploit the relationship between the estimated fundamental frequency, f_0, and the period (in seconds), T_0. We will note from Fig. 7.2 that the estimated frequency at $n = 5350$ is approximately 217.241 Hz. Therefore, the approximate fundamental period is $T_0 = 1/217.241$ sec. Now, we need to determine the number of samples contained within a 1/217.241 sec. period. Since we know the sampling period of the instrument recording, we can compute this value by dividing the number of samples per second, f_s, by the fundamental frequency, f_0. To ensure that N is an integer number, we must round the final result. Consequentially, we can compute the total number of samples as:

$$N = \text{round}(f_s/f_0)\,.$$

In your `piano_slice.m` script, add the following lines:

Listing 7.2: `piano_slice.m` (continued)

```
11 F0 = 217.241; % from the plot_pitch Pitch Graph
12 n_a = 5350;  % from the plot_pitch Amplitude/Pitch Graphs
13 N = round(Fs/F0);
```

```
14
15 x_slice = cut_sample(x, n_a, N);
```

Now, we need to verify that we obtained the proper slice. We can do so by plotting three repetitions of the slice and inspecting the resultant graph for issues. For example, if we identified a region of the instrument recording waveform that was not sufficiently flat (our n_a is wrong), or the estimated fundamental frequency, f_0, is incorrect, we may see sharp jumps at the boundary of each repetition, as in Fig. 7.4.

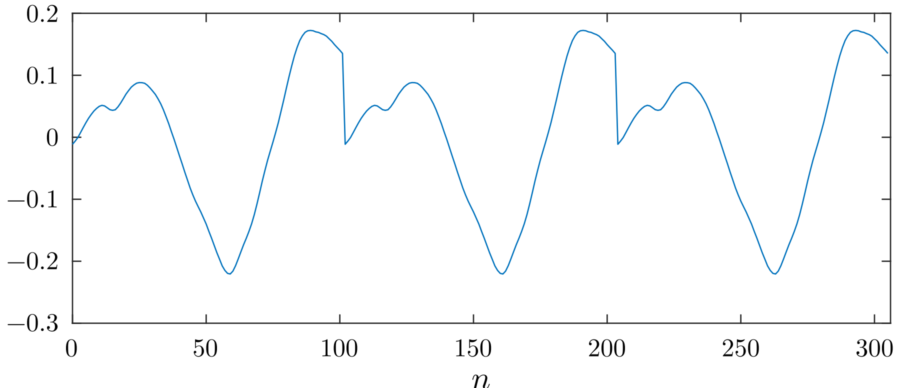


Figure 7.4: An example of *three* repetitions of an improperly isolated instrument *slice*; *note the sharp edge at the peaks*—indicative of an improper estimation of the fundamental frequency, f_0, *or* a region of the waveform where the amplitude envelope is not sufficiently flat.

Add the following code to your `piano_slice.m` script:

Listing 7.3: `piano_slice.m` (continued)

```
17 % Plot three periods of x_slice to verify it looks as we expect
18 figure(2);
19 x_slice_repeat = repmat(x_slice, 3, 1);
20 plot(x_slice_repeat)
```

Run the MATLAB script. You should see a result similar to that depicted in Fig. 7.5.

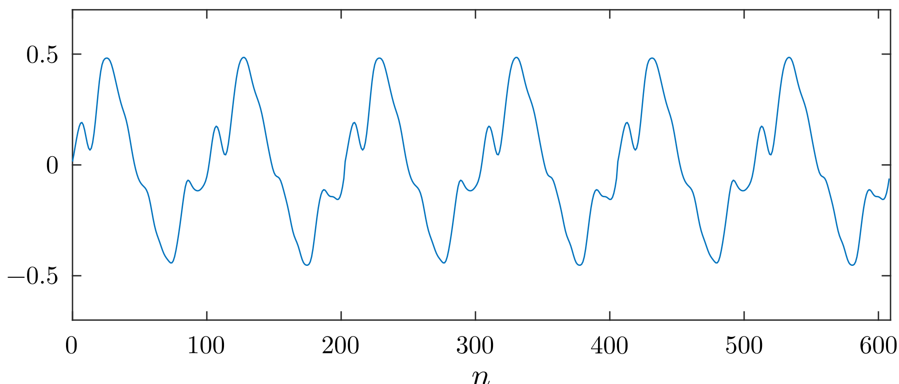


Figure 7.5: An example of three (yes *three*, not six) repetitions of an instrument slice whose fundamental frequency, f_0, was improperly estimated. In this case, we see *six* periods of a periodic waveform when we had hoped to see *three*. Hence, the estimated f_0 is *half* of its *true* value.

What happened? Why do we see six periods rather than three? This is a common problem with the pitch estimation algorithm MATLAB uses—it often lowballs the estimated frequency by an integer multiple. In this case, we see twice as many periods as we expect. Therefore, our estimated `F0` is half the size it needs. Return to line 11 of Listing 7.2 and multiply our `F0` value by two:

```
11  F0 = 2*217.241; % from the plot_pitch Pitch Graph
```

Run the MATLAB script. You should see a result similar to that depicted in Fig. 7.6. This tells us that we have successfully isolated our slice.

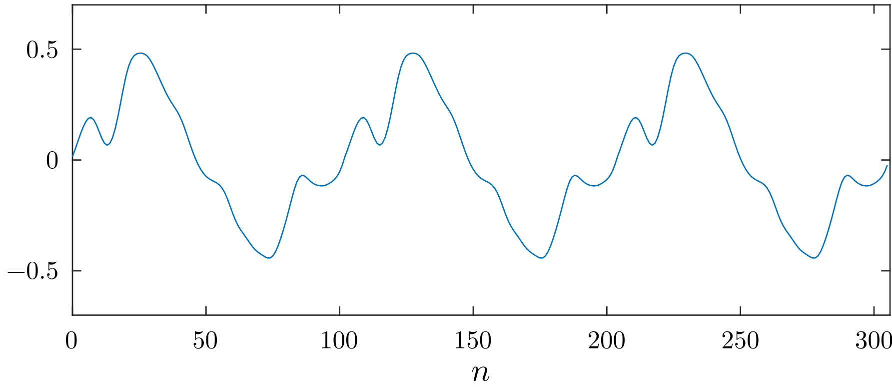


Figure 7.6: Three repetitions of a properly isolated instrument slice derived from `piano.wav`.

7.4.2 Re-synthesizing the Instrument

Our next order of business is estimating the Fourier Series coefficients, X_k, for $-21 \leq k \leq 21$ for our instrument slice. Remember, we are estimating the coefficients of a single period of our slice, not the version that was replicated three times. This can be done trivially using the `find_Xks(...)` function (refer to the previous lab exercise for usage if needed).

Add the following to your `piano_slice.m` script:

Listing 7.4: `piano_slice.m` (continued)

```
23 % Compute the FS coefficients for a *single* period of x_slice
24 K = 21;
25 X_ks = find_Xks(x_slice, K, 1/Fs);
26 k = -K:K;
```

Using your knowledge of MATLAB, update your `piano_slice.m` script to generate a figure containing two subplots depicting the magnitude and phase (in degrees) of the Fourier Series coefficients `X_ks`. Your figure should appear similar to Fig. 7.7. Include this figure in your report.

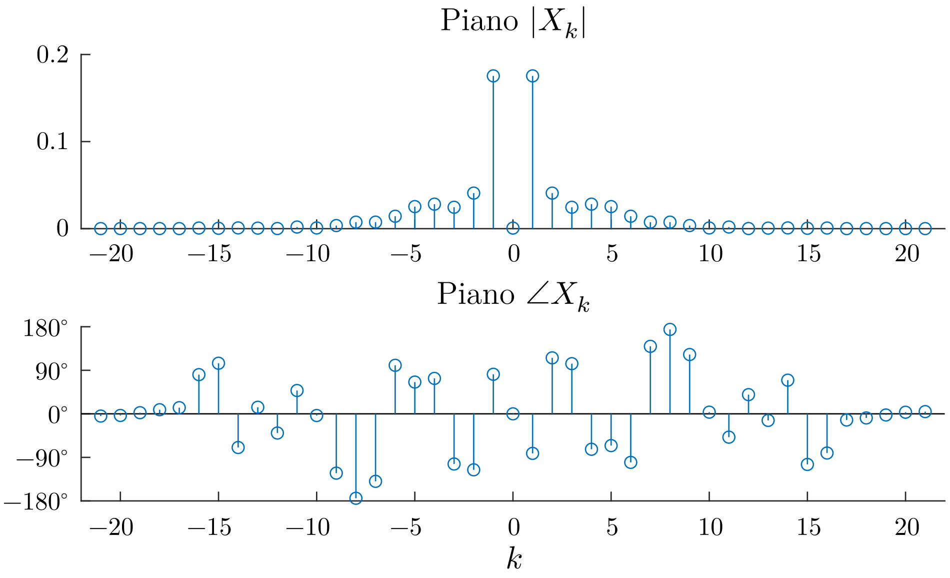


Figure 7.7: The magnitudes and phases (in degrees) of the FS coefficients, X_k, of the instrument slice derived from `piano.wav`.

Using your knowledge of MATLAB and the `fsynth(...)` function, update your `piano_slice.m`

script to re-synthesize a single period of the piano with a period of `T0=1/F0` (use the same `Fs` as the original instrument recording). *Hint*: Remember that the `T_dur` parameter to `fsynth(...)` is specified in seconds, not number of samples. Generate a figure that overlays the re-synthesized signal atop the original slice. Your figure should appear similar to Fig. 7.8. Include this figure in your report.

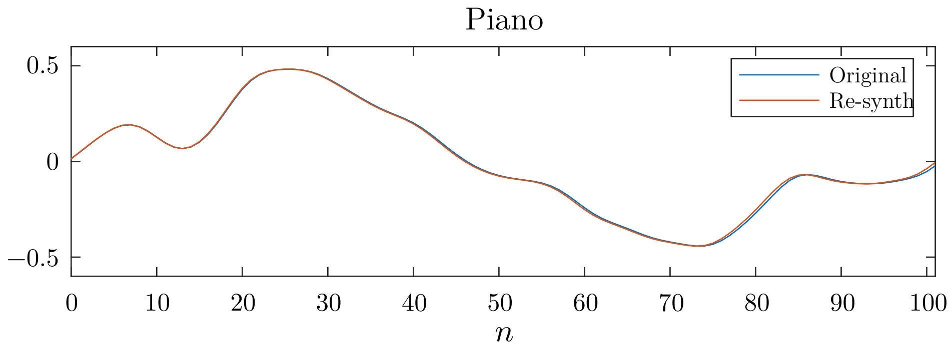


Figure 7.8: Comparison between a *single period* of the original slice from `piano.wav` and the re-synthesized slice.

Using your knowledge of MATLAB and the `fsynth(...)` function, update your `piano_slice.m` script to synthesize *two* seconds of the piano sound at `T0=1/F0` (use the same `Fs` as the original instrument recording). Save this signal to a `.wav` file called `piano_resynth.wav`. Include `piano_resynth.wav` as an attached file in your submission.

7.4.3 Re-synthesize two more Instruments

Signal #	File Name	f_s	Pitch (f_0)	Starting Sample of Cut (n_a)	Number of Samples (N)
1	`piano.wav`	44.1 kHz	434.482 Hz	5350	102
2					
3					

Table 7.1: Properties of extracted instruments.

Repeat the process you went through for `piano_slice.m` for two other instruments of your choice. For each instrument, create a new script (e.g., `oboe_high_slice.m`), and fill out Table 7.1 with the filenames, sampling rates (f_s), f_0, n_a and N values. Include the magnitude/phase figure

and re-synthesized vs. original slice figures in your report for each instrument. Also, include the *two* second re-synthesized audio files for each instrument as attached files in your submission.

7.4.4 Synthesize a Chromatic Scale

Note	Frequency (Hz)
A_4	440.00
$A^\sharp{}_4$	466.16
B_4	493.88
C_5	523.25
$C^\sharp{}_5$	554.37
D_5	587.33
$D^\sharp{}_5$	622.25
E_5	659.25
F_5	698.46
$F^\sharp{}_5$	739.99
G_5	783.99
$G^\sharp{}_5$	830.61
A_5	880.00

Table 7.2: Note frequencies of an A-Chromatic scale; reproduced from Suits.[5]

Using your knowledge of MATLAB, select one of the three instruments you re-synthesized to synthesize the notes of the A-Chromatic scale given in Table 7.2. Each whole note (notes without the $\sharp$) should be one second long, while sharp ($\sharp$) notes should be 0.5 seconds long. Save this scale to a file called `chrom_scale.wav`. Include `chrom_scale.wav` as an attached file in your submission. *Hint*: Refer to the previous lab exercise where you synthesized a C-Major scale. Use the same technique of successive array concatenation.

7.5 Report Checklist

Work with lab partner(s) to create a lab report for this lab exercise. Be sure to include the names of each person in your group on the report. This report should be a typed document that outlines the basic procedure of the lab exercise and should include any results you were asked to obtain. Your report should follow a *cohesive narrative*—it should always be apparent *why* you

have presented the results you have. Do not simply insert results without (1) motivating the problem and (2) providing an analysis of the data. Ensure that your report is grammatically correct with complete sentences. This does not mean your report must be wordy with filler—in fact, avoid filler and state your ideas concisely.

Be sure the following are included in your report:

- Section 7.2: Give the MATLAB command (leveraging `repmat(...)`) to initialize a matrix, **C**, which consists of the matrix **A** from Eq. (7.1) repeated *one hundred* times *vertically*.
- Section 7.4.2: Update `piano_slice.m` to generate a figure containing two subplots depicting the magnitude and phase, similar to that depicted in Fig. 7.7. Include this figure in your report.
- Section 7.4.2: Update `piano_slice.m` to re-synthesize a single period of the piano. Generate a figure that overlays the re-synthesized signal atop the original slice (as in Fig. 7.8). Include this figure in your report.
- Section 7.4.2: Update `piano_slice.m` to generate *two* seconds of the piano sound and save it to a file, `piano_resynth.wav`. Include `piano_resynth.wav` as an attached file in your submission.
- Section 7.4.3: Repeat the process you went through for `piano_slice.m` for two other instruments. Fill in the data for Table 7.1. For each non-piano instrument, include the magnitude/phase figures, re-synthesized vs. original figures in your report. Also, include both re-synthesized audio files for each instrument as attached files in your submission.
- Section 7.4.4: Choose an instrument to synthesize the A-Chromatic scale (given in Table 7.2). Each whole note should be 1 second long, while each sharp (♯) note should be 0.5 seconds long. Save this synthesized scale to the file `chrom_scale.wav` and include it as an attached file in your submission.

7.6 Acknowledgements

This lab exercise is based on an exercise drafted by **Taylor Burton** in co-operation with **Andrew Bolstad**.

7.A Plot Pitch Function

Listing 7.5: `plot_pitch.m`

```
function plot_pitch(x, Fs, plotTitle)
% PLOT_PITCH plots the estimated fundamental pitch/frequency of a signal.
%   PLOT_PITCH(x, Fs, plotTitle) plots the time-domain signal, x,
%   sampled at the rate Fs alongside its estimated fundamental frequency
%   with respect to time. The plot title is specified by plotTitle.
%
% Inputs
%   x: a vector containing a time domain signal.
%   Fs: a scalar specifying the sampling rate of x.
%   ploTitle: a character array specifying a title for the resulting plot.

% Written By: Julie Dickerson and Andrew Bolstad, 2020
% Updated By: Aaron Fonseca, 2024


    [f0, idx] = pitch(x, Fs);

    axTop = subplot(2,1,1);
    plot(x);
    ylabel('Amplitude (x[n])');
    title(plotTitle);

    axBottom = subplot(2,1,2);
    plot(idx, f0);
    ylabel('Pitch (Hz)');

    linkaxes([axTop, axBottom], 'x')
    xlabel('Sample Number (n)');
end
```

7.B Cut Sample Function

Listing 7.6: `cut_sample.m`

```
function x_slice = cut_sample(x, n_a, N)
% CUT_SAMPLE cuts/slices the sampled signal x.
%   x_slice = CUT_SAMPLE(x, n_a, N) cuts/slices the sampled signal x at
%           the offset n_a for N samples.
%
% Inputs
%   x: a vector containing a time domain signal.
%   n_a: a positive integer specifying the offset (in samples) to begin
%       the cut/slice.
%   N: a positive integer specifying the total number samples in the
%      resulting cut/slice.

% Written By: Julie Dickerson and Andrew Bolstad, 2020
% Updated By: Aaron Fonseca, 2024



   x_slice = x(n_a:(n_a+N)-1, :);
end
```

7.C Compute Fourier Series Coefficients Function

This function requires the `find_Xk.m` function (see Appendix 7.D) to run.

Listing 7.7: `find_Xks.m`

```
function X_ks = find_Xks(x, largestK, Ts)
% FIND_XKS Computes the Fourier series coefficients of x between +/-largestK.
%   X_ks = FIND_XKS(x, largestK, Ts) computes the Fourier seires coefficients
%   between -largestK and largestK of the signal x, where largestK is a
%   positive integer, x is a column vector containing a *single* period of a
%   function and Ts is the sampling period of x.
```

```

% Written By: Julie Dickerson, 2020
% Updated By: Aaron Fonseca, 2024


    % Set up the coefficient index vector
    ks = -largestK:largestK;

    % setup vector for results
    numK = length(ks);
    X_ks = zeros(size(ks));

    % loop through all coefficients to compute coefficients
    for i = 1:numK
        X_ks(i) = find_Xk(x, ks(i), Ts);
    end
end
```

7.D Compute Fourier Series Coefficient Function

Listing 7.8: `find_Xk.m`

```
function X_k = find_Xk(x, k, Ts)
% FIND_XK Computes a single Fourier series coefficient via Riemann approx.
%   X_k = FIND_XK(x, k, Ts) computes the kth Fourier series coefficient of the
%   signal: x, where x is a column vector containing a *single* period of a
%   function and Ts is the sampling period of x.

% Written By: Julie Dickerson, 2020
% Updated By: Aaron Fonseca, 2024


    % Ensure that x is a column vector
    [rx, ] = size(x);
```

```
13      if (rx==1)
14          x=x.';
15      end
16
17      % Compute the length of the period, T.
18      T = length(x) * Ts;
19      w0 = (2*pi) / T;
20
21      t = 0:length(x)-1;
22      t = t * Ts;
23
24      % Compute the complex part over all values of t.
25      basis_func = exp(-1j*k*w0*t);
26
27      % The complex part mulitiplied by the function yields the Riemann sum.
28      X_k = (Ts/T) * basis_func*x;
29  end
```

7.E Synthesize Fourier Series Function

Listing 7.9: `fsynth.m`

```
1   function x_hat = fsynth(X_ks, T0, Ts, Tdur)
2   % FSYNTH Synthesizes a time-domain signal given Fourier series coefficients.
3   %   x_hat = FSYNTH(X_ks, T0, Ts, Tdur) synthesizes 'Tdur' seconds of a T0
4   %   periodic signal, with a sampling period Ts, specified by the Fourier
5   %   series coefficients given in X_ks.
6   %
7   % Inputs
8   %   X_ks: a 2K+1 length vector of the form:
9   %            [X[-K] ... X[-1], X[0], X[1] ... X[K]]
10  %        where X[k] gives the kth Fourier series coefficient of the signal
11  %        to synthesize.
12  %   T0: a scalar specifying the fundamental period of the signal to synthesize.
```

```
%  Ts: a scalar specifying the sampling period of the signal to synthesize.
%  Tdur: a scalar specifying the total duration (in seconds) of the signal to
%       synthesize.

% Written By: Julie Dickerson, 2020
% Updated By: Aaron Fonseca, 2024


    if rem(numel(X_ks), 2)  = 1
        error('numel(X_ks) = %d. The length of the X_ks vector should be odd.', numel(X_ks))
    end
    K = floor((numel(X_ks) - 1) / 2);
    ks = -K:K;

    % Find omega wich is (2*pi)/Period
    % Frequency is the inverse of the Period
    % T0 is the fundamental period of the function
    w0 = (2*pi)/T0;

    % make time vector for evaluation
    t = 0:Ts:Tdur;

    % Compute the Fourier approximation
    x_hat = real(X_ks * exp(1j*w0*ks'*t));
end
```

7.R References

[1] "Repeat copies of array - MATLAB," MathWorks Inc. (2023), [Online]. Available: https://www.mathworks.com/help/matlab/ref/repmat.html.

[2] M. T. Group. "Good-sounds: Good-sounds.org dataset," Universitat Pompeu Fabra. (2015), [Online]. Available: https://www.upf.edu/web/mtg/good-sounds.

[3] TEDAgame. "Pack: 88 piano keys, long reverb," Freesound.org. (2015), [Online]. Available: https://freesound.org/people/TEDAgame/packs/25405/.

[4] "Audio toolbox - MATLAB," MathWorks Inc. (2023), [Online]. Available: https://www.mathworks.com/products/audio.html.

[5] B. H. Suits. "Physics of music notes, musical note frequencies for an equal tempered scale," Physics Department, Michigan Technological University. (2023), [Online]. Available: http://pages.mtu.edu/~suits/notefreqs.html.

Introduction to Digital Images

An introduction to digital imaging in MATLAB

Overview

This lab exercise introduces digital images as a new higher-dimensional signal type. Students will use the `level_slice.m` and `contrast_stretch.m` MATLAB scripts (provided in this exercise's *supplementary materials* or in Appendices 8.A–8.B) to manipulate and enhance gray-scale images.

Learning Objectives

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

1. Display images in MATLAB.
2. Manipulate image pixel values in MATLAB.
3. Perform level-slicing and contrast-stretching operations on images in MATLAB.

8.1 Introduction

In this lab exercise, you will be introduced to digital images as a higher-dimensional signal type. You will explore image histograms and use MATLAB to enhance and manipulate images.

8.2 Images in MATLAB

8.2.1 Monochromatic Images

A monochromatic digital image is a two-dimensional array of pixels (short for picture element) with a given width, w, and height, h. Images can be represented as a matrix $\mathbf{A}$:

$$\mathbf{A} = \begin{bmatrix} a_{(1,1)} & a_{(1,2)} & \cdots & a_{(1,\,w-1)} & a_{(1,w)} \\ a_{(2,1)} & a_{(2,2)} & \cdots & a_{(2,\,w-1)} & a_{(2,w)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{(h-1,1)} & a_{(h-1,2)} & \cdots & a_{(h-1,\,w-1)} & a_{(h-1,w)} \\ a_{(h,1)} & a_{(h,2)} & \cdots & a_{(h,\,w-1)} & a_{(h,w)} \end{bmatrix}$$

where each element in the matrix, $a_{(y,\,x)}$ represents the brightness value of that particular pixel—where y specifies the vertical position of a specific pixel (i.e., the *row* within $\mathbf{A}$) and x specifies the horizontal position of a particular pixel (i.e., the *column* within $\mathbf{A}$).

Monochromatic images are displayed using black and white and shades of gray, so they are also called *gray-scale* images. In this lab exercise, we will consider only sampled gray-scale images. To encode each element $a_{(y,\,x)}$ within $\mathbf{A}$, an 8-bit, unsigned, integer representation (`uint8`) is typically used. With unsigned 8-bit integers, the maximum value would be $2^8 - 1 = 255$, and there would be $2^8 = 256$ different gray levels for the display, from 0 to 255.[1][2] In such images, 0 corresponds to a completely black pixel while 255 corresponds to a completely white pixel. An illustration of pixel intensities encoded by an example matrix, $\mathbf{A}$, is given in Fig. 8.1. Here, the values the values of each pixel $a_{(y,\,x)}$ in $\mathbf{A}$ are printed in white.

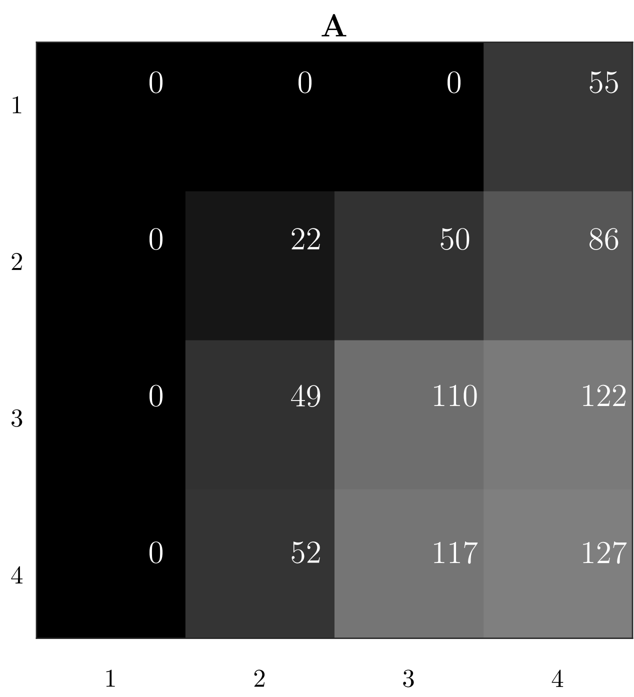


Figure 8.1: An illustration of an example image given by the 4×4 matrix $\mathbf{A}$; the corresponding pixel values/intensities are given by the values printed in white.

8.2.2 Displaying and Exporting Images

Most of the checkpoints in this lab exercise will require you to produce one or more output images. These will need to be incorporated into your report.

Reading Images

You can load an image, `myimage.jpg`, into MATLAB's **Variable Workspace** using the `imread(...)` function:

```
A = imread('myimage.jpg');
```

This will produce an image matrix, `A`, of data type `uint8`.

Displaying Images

You can display the image array `A` with the following commands:

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

If A is a gray-scale image, the colormap(...) function is needed to tell MATLAB which color to display for each possible pixel value.

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, A, to a file using the imwrite(...) function:

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

Note that if A is of type uint8, the imwrite(...) function assumes a dynamic range of $[0, 255]$, and will clip any values outside that range. If A 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, enter the command:

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

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

```
A_rgb = imread('peppers.png');
imshow(A_rgb)
A = rgb2gray(A_rgb);
figure;
imshow(A)
```

8.2.3 An Image of the Moon

Open MATLAB and run the following command:

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

The file, `moon.tif`, is an example image that comes packaged with the MATLAB image processing toolbox. You should still be able to load it even though it is not explicitly present in your **Current Folder View**.

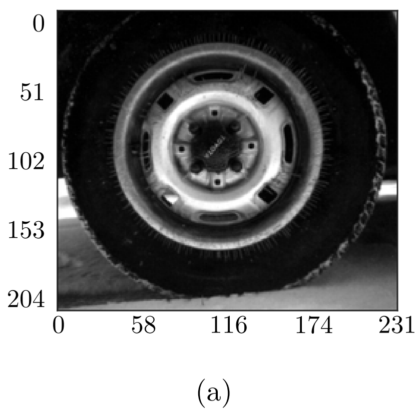
1. What is the size of the image example? $N \times M$ pixels?
2. What is the variable type? (*Hint:* You can view variable types in the **Variable Workspace**.)
3. Write out one line of MATLAB code to see the intensity value of any given pixel in the image. (*Hint:* How do you index values within a matrix, $\mathbf{A}$?)

8.3 Image Manipulation in MATLAB

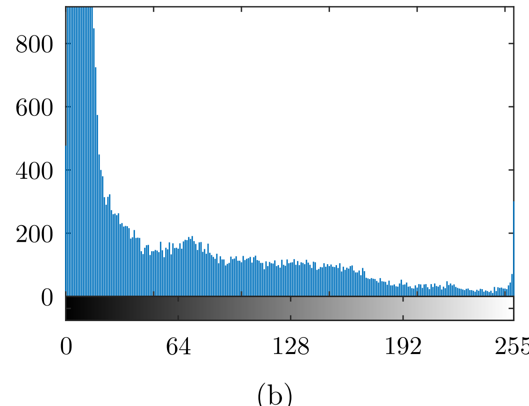
The goal of image manipulation is to improve the image in ways that increase the performance of a system for human viewing or computer vision applications.

8.3.1 Image Histograms

Image histograms are a simple but useful method to analyze images and enhance the quality of the image (at least for a human observer) by performing point transformations. In gray-scale images, the histogram looks at the distribution of the intensity values of the image. For each range of gray-scale values, the number of pixels falling in that range is 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. Fig. 8.2 shows the MATLAB image `tire.tif` and its histogram. Note that most of the pixels have low values in the range of 0–25 in intensity.



(a)



(b)

Figure 8.2: Image of a tire with a histogram of intensity values. Most of the pixels have low intensities.

8.3.2 Contrast Stretching and Intensity Level Slicing

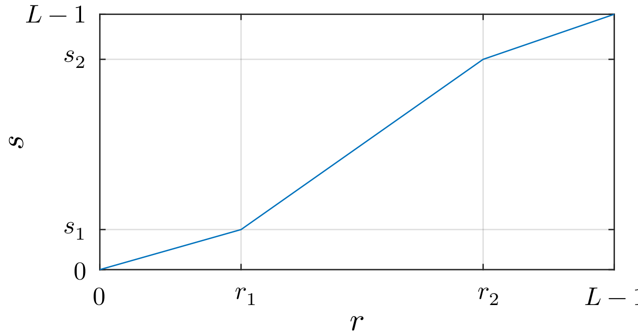
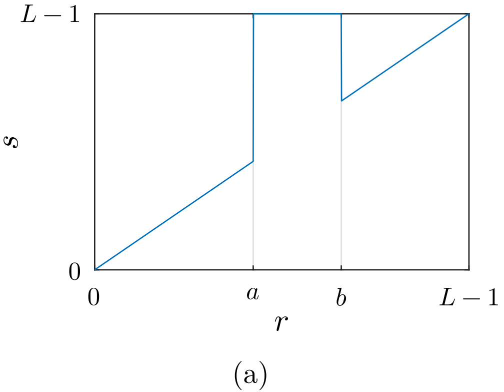
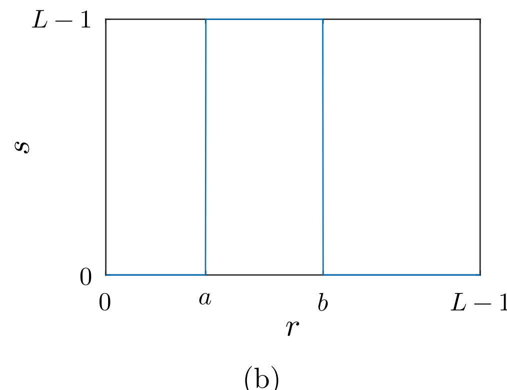


Figure 8.3: Example contrast mapping $s = T(r)$. The function T is applied to each pixel. Inspired by a figure from Gonzalez and Woods.[2]

Low-contrast images result from poor illumination or lack of response from the imaging sensor. Contrast stretching aims to increase the dynamic range of the gray levels in the image being processed. This is implemented by a mapping function, $s = T(r)$. Fig. 8.3 shows a typical transformation for contrast stretching. The locations of points (r_1, s_1) and (r_2, s_2) control the shape of the contrast function. If $r_1 = r_2$ and $s_1 = s_2$, then the transformation is a linear function. If $r_1 = r_2$, $s_1 = 0$, and $s_2 = L - 1$, then the transformation is a thresholding function. Intermediate values produce different types of spread. The mapping is a piecewise linear interpolation.[2][3]



(a)

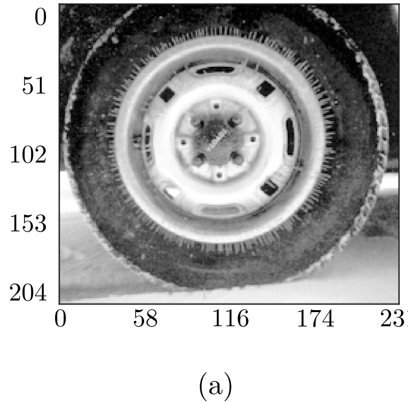


(b)

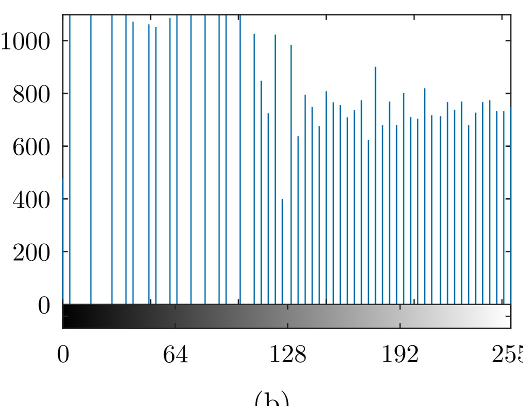
Figure 8.4: Example mapping functions for *intensity level slicing*. The mapping on the left brightens pixels with intensities between a and a (i.e., $a \leq r \leq b$). The mapping on the right performs the same brightening procedure but also zeros out all pixels with intensities below a and above b. Inspired by figures from Gonzalez and Woods.[2]

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: brighten the pixels in the range and leave the other pixels the same, or brighten the pixels in the range and zero out the others. Levels can be deleted using a similar approach. Examples are shown in Fig. 8.4.

8.3.3 Histogram Equalization



(a)



(b)

Figure 8.5: Tire image after histogram equalization with a histogram of its intensities.

The histogram equalization operation tries to change the pixel distribution to be as close as possible to a uniform distribution. All gray levels from 0 to $2^L - 1$ (where L 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 Fig. 8.5 for the `tire.tif` image. Compare this to the original image in Fig. 8.2. 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)
```

8.4 Isolating Areas of Interest

The *supplementary materials* for this lab exercise provide MATLAB files: `level_slice.m` and `contrast_stretch.m` (these are also provided in Appendices 8.A–8.B). These files are not stand-alone MATLAB scripts. You cannot run them by clicking the **Run** button in MATLAB. Rather, these are custom MATLAB functions. You can now *call* and use the `help` command on them as you would any other function in MATLAB—provided they are visible within your **Current Folder View**. Create a new folder that will serve as your project directory, and place these scripts in that folder.

1. Navigate to the Iowa State Entomology Image Gallery's page on periodical cicadas[4] and save the close-up image. The images on this site have been made available for educational purposes only and should not be used in any commercial endeavors. In your lab report, provide a link to the webpage you downloaded the image from and provide proper credit to the photographer and the Iowa State Entomology Department. Rename the downloaded image as `cicada.jpg`. Create a new script, `cicada_process.m` which loads `cicada.jpg` into MATLAB, converts it to gray-scale and computes its histogram. Include the histogram

and image in your lab report.

(a) What is the range of the pixel intensity values in the original image?

(b) In your `cicada_process.m` script, use any combination of contrast stretching, level intensity slicing, and histogram equalization to increase the range of the image intensity. Describe and show the results of the operations in your lab report. Give details on the ranges for each routine. Does one method work better than the other? Explain your answer.

2. Navigate to the Science and Plants for Schools photograph of Broad Bean mitosis[5] and save the image. This particular image has been made available for non-commercial purposes. In your lab report, provide a link to the webpage you downloaded the image from and provide proper credit to the photographer. Rename the downloaded image as `cell1.jpg`. Create a new script `cell_process.m` which loads `cell1.jpg` into MATLAB, and converts the image to gray-scale. Use the methods described in Sec. 8.3 to isolate the nuclei of the cells in the image 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 cells in the slide using their nuclei. In your lab report, describe your steps 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.

8.5 Report Checklist

Work with lab partner(s) to create a lab report for this lab exercise. Be sure to include the names of each person in your group on the report. This report should be a typed document that outlines the basic procedure of the lab exercise and should include any results you were asked to obtain. Your report should follow a *cohesive narrative*—it should always be apparent *why* you have presented the results you have. Do not simply insert results without (1) motivating the problem and (2) providing an analysis of the data. Ensure that your report is grammatically correct with complete sentences. This does not mean your report must be wordy with filler—in fact, avoid filler and state your ideas concisely.

Be sure the following are included in your report:

- Section 8.2.3: Answer the three questions about `moon.tif`.
- Section 8.4: Include `cicada.jpg` and its histogram. Provide a link to the image source and properly credit the photographer and the ISU Entomology Department.
- Section 8.4: Give the range of pixel intensities for `cicada.jpg`.
- Section 8.4: Increase the range of the image intensity for `cicada.jpg` using any combination of contrast stretching, level intensity slicing, and histogram equalization. Does one method work better than the other? Explain your answer.
- Section 8.4: Create A MATLAB script to isolate the nuclei in `cell1.jpeg` using any combination of contrast stretching, level intensity slicing, and histogram equalization. Provide a link to the image source and properly credit Science and Plants for Schools. Include your script and describe the stages of your processing pipeline. Include images of the intermediate results after each stage in your pipeline.

8.A Level Slice Function

Listing 8.1: `level_slice.m`

```
function Y = level_slice(X, a, b, sliceType)
% LEVEL_SLICE Performs intensity level slicing on grayscale image.
%   Y = LEVEL_SLICE(X, a, b, sliceType) returns an image, Y, where
%   the pixel values between a and b (inclusive) are set to 255 (white).
%   Pixels not set to 255 either remain unchanged (default behavior) or
%   are set to 0 if sliceType is set to 'zero'.
%
% Inputs
%   X: a uint8 matrix representing a grayscale image.
%   a: an integer representing the lower bound of pixel values to set to white.
%   b: an integer representing the upper bound of pixel values to set to white.
%   sliceType: a character array specifying the type of slicing operation. If
%              set to 'zero', all pixels not set to white will be set to 0
%              (black). If set to any other string, the pixels not set to white
%              remain unchanged.

```

```
% Written By: Andrew Bolstad, 2022
% Updated By: Aaron Fonseca, 2024


    % convert X to double for processing
    X = double(X);

    Idx = X >= a & X <= b;
    X(Idx) = 255;

    % check if sliceType == 'zero'
    if strcmp(lower(sliceType), 'zero')
        X( Idx) = 0;
    end

    % place updated X into Y and convert to uint8
    Y = uint8(X);
end
```

8.B Contrast Stretch Function

Listing 8.2: `contrast_stretch.m`

```
function Y = contrast_stretch(X, p1, p2)
% CONTRAST_STRETCH performs contrast stretching on a grayscale image.
%   Y = CONTRAST_STRETCH(X, p1, p2) applies contrast stretching to each pixel
%   in the image X. Contrast stretching uses a piecewise linear mapping of
%   input pixels to output pixels. The mapping consists of three linear
%   segments:
%       1. (0,0) to (r1,s1)
%       2. (r1,s1) to (r2,s2)
%       3. (r2,s2) to (255,255)
%   where p1 = [r1, s1] defines the first point and p2 = [r2,s2] defines
%   the second point.
```

```
12  %
13  % Inputs
14  %   X: a uint8 matrix representing a grayscale image.
15  %   p1: a two-element vector of the form [r1,s1]
16  %   p2: a two-element vector of the form [r2,s2]
17
18  % Written By: Andrew Bolstad, 2022
19  % Updated By: Julie Dickerson, 2023
20  % Updated By: Aaron Fonseca, 2024
21
22
23  % Convert image to double for processing.
24      X = double(X);
25
26      % This isn't totally necassary, but it makes the code more human-readable
27      % with very little overhead.
28      r1 = p1(1);
29      s1 = p1(2);
30      r2 = p2(1);
31      s2 = p2(2);
32      M = 255;
33
34      if r2 < r1
35          error('r2 < r1, this is not permitted for this mapping')
36      end
37      if s2 < s1
38          error('s2 < s1, this is not permitted for this mapping')
39      end
40
41      Y = zeros(size(X));
42      if r1 > 0
43          m1 = s1/r1;
44          seg1_eq = @(r) (r < r1).*(m1.* r);
45          Y = Y + seg1_eq(X);
46      end
47      if r2 > r1
```

```
48        m2 = (s2-s1) / (r2-r1);
49        b2 = s1 - (m2*r1);
50        seg2_eq = @(r) (r >= r1 & r <= r2).*(m2.*r + b2);
51        Y = Y + seg2_eq(X);
52    elseif r1 == r2
53        Y = Y + (X == r2).*(s2);
54    end
55    if M > r2
56        m3 = (M-s2) / (M-r2);
57        b3 = s2 - (m3*r2);
58        seg3_eq = @(r) (r > r2).*(m3.*r + b3);
59        Y = Y + seg3_eq(X);
60    end
61    Y = uint8(Y);
62
63 end
```

8.R References

[1] J. Jackson. "Introduction to digital image processing [PowerPoint slides]," University of Alabama. (Jul. 2018), [Online]. Available: http://web.archive.org/web/20180728062023/http://jjackson.eng.ua.edu/courses/ece482/lectures/LECT01-2.pdf.

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

[3] J. Jackson. "Intensity transformations and spatial filtering [PowerPoint slides]," University of Alabama. (Jul. 2018), [Online]. Available: http://web.archive.org/web/20180728062044/http://jjackson.eng.ua.edu/courses/ece482/lectures/LECT05-2.pdf.

[4] S. Frommelt. "Periodical cicada (closeup)." (2000), [Online]. Available: https://www.ent.iastate.edu/imagegal/homoptera/cicada/17yrcicada.html.

[5] Science and P. for Schools. "Mitosis in root tip of broad bean (vicia faba) 1/2." (2012), [Online]. Available: https://www.flickr.com/photos/75759067@N07/galleries/72157714745770573/.

Digital Image Processing

Filtering Digital Images in MATLAB

Overview

In this lab exercise, students will use image filtering routines in MATLAB to process a selection of images provided in this exercise's *supplementary materials*.

Learning Objectives

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

1. Classify regions of images in terms of their spatial frequency.
2. Compute the spatial period of simple, oscillating images.
3. Manually compute the per-pixel result of a sliding window filter.
4. Apply sliding window masks to source images using MATLAB's imfilter function.
5. Initialize N-dimensional moving-average masks in MATLAB.
6. Assess the efficacy of various filter types to attenuate Gaussian and 'salt and pepper' noise.
7. Implement and apply the Canny edge detection algorithm in MATLAB.

9.1 Introduction

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

9.2 Background

9.2.1 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 discontinuities, the approach is partitioning an image based on abrupt changes in gray level. Objects of interest are isolated points, lines, and edges.

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. The figure below 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 when 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.

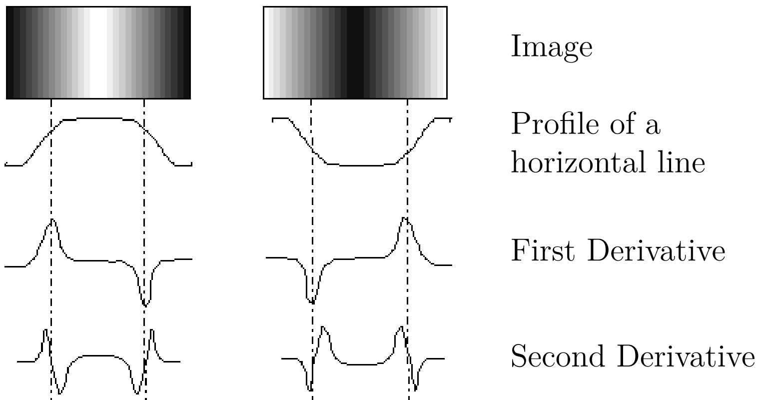


Figure 9.1: Synthetic images with horizontal pixel intensity, first derivative of intensity, and second derivative of intensity; reproduced from Gonsazlez and and Woods.[1]

9.2.2 Regions of Spatial Frequency

For Fig. 9.2, explain which part of the image has the lowest spatial frequency: specify a corner, side, or middle of the image.

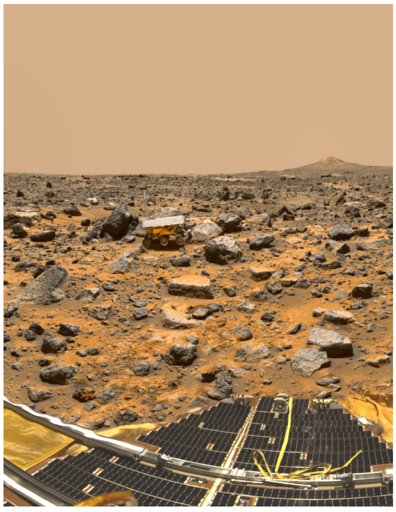


Figure 9.2: The surface of Mars, as seen from the Pathfinder lander; reproduced from NASA.[2]

9.2.3 Computing Spatial Period

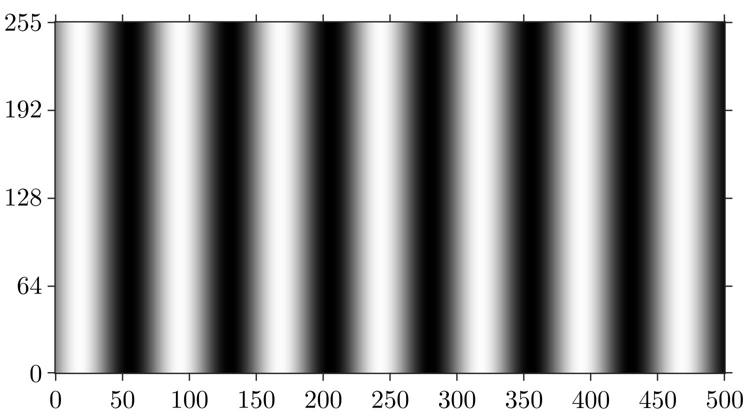


Figure 9.3: Horizontal Oscillation.

Compute the image's horizontal spatial *period* of Fig. 9.3 in terms of pixels/cycle. Give a numerical answer.

9.2.4 Sliding Window Filters

One way of filtering is to filter the rows with a one-dimensional filter and then filter the columns with a one-dimensional filter. Two-dimensional filters can also be used that use the neighborhood of pixels 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.

The operations occur 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 *filter mask*. The use of *spatial masks* (or *kernels*) 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 3×3 linear filter can be written in matrix form as:

$$\mathbf{H}_3 = \begin{bmatrix} h_{(1,1)} & h_{(1,0)} & h_{(1,\text{-}1)} \\ h_{(0,1)} & h_{(0,0)} & h_{(0,\text{-}1)} \\ h_{(\text{-}1,1)} & h_{(\text{-}1,0)} & h_{(\text{-}1,\text{-}1)} \end{bmatrix}.$$

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

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

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, as shown in the figure below. Low-pass (LP) filters attenuate

(or eliminate) high-frequency components characterized by edges and sharp details in an image with the net effect of blurring the image. High-pass (HP) filters attenuate (or eliminate) low-frequency components such as slowly varying characteristics. The overall effect is a sharpening of edges. Band-pass (BP) filters attenuate (or eliminate) a given frequency range. This is primarily used for image restoration.[1]

9.2.5 Manually Computing the Value of a Filtered Pixel

Given the filter mask, **H**, in Fig. 9.4a and the intensity values for the image matrx depicted in Fig. 9.4b, manually compute the filter output for the pixel at location (3, 3).

$\mathbf{H} =$

-1	0	1
-2	0	2
-1	0	1

(a) The contents of the filter mask, **H**; the gray cell indicates the location of the origin, (0, 0), of the mask.

	1	2	3	4	5	6
1	5	25	25	30	20	10
2	20	20	30	60	25	30
3	35	30	40	70	40	40
4	80	80	70	70	60	50

(b) The contents of an image which the filter mask, **H**, is being applied; the gray cell indicates the pixel, (3, 3) at which the filter is being applied.

Figure 9.4: Convolution of a filter mask, **H**, with an image for the pixel at (3, 3).

9.2.6 MATLAB Background

This lab exercise will make heavy use of the MATLAB routine `imfilter(...)`. In the expression: `Y = imfilter(X, H);`, `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 3×3 averaging filter is given below:

```
X = imread('moon.tif');
H = ones(3, 3) / 9;
Y = imfilter(X, H);
```

There are two things to remember with `imfilter(...)`. First, it returns an image in the same

format as the input. If the function gets a non-integer value for a uint8 image, it rounds it to the closest integer. Second, it clips pixel values outside the range $[0, 255]$ to 0 or 255. These non-linear operations affect 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 behavior of imfilter(...) is to assume a black border. Details can be found by reading about padding options on the help page.

9.3 Filtering Images

This portion of the assignment uses image_filt.mat, which is provided in this exercise's *supplementary materials*. Create a new folder to use as your project directory and place image_filt.mat within it. Create a new script in your project directory. In this script, enter the command to load image_filt.mat. (Refer to previous lab exercises for information on loading .mat files.) Run your script. You should see three new variables in your **Variable Workspace**: I, I_g, and I_sp. These variables each contain matrices representing gray-scale images.

For each checkpoint in this lab exercise, you will be asked to display "*before-and-after*" filtering results. Suppose you have just completed a checkpoint and produced a filtered version of the image, I, called: filtI. There is a convenient function that will allow you to display the original image, I, alongside the filtered result, filtI, without using subplot(...) commands. This function is:

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

You may also need to display a close-up of parts of the image that illustrate the filter effect well. To do this, it helps display the image with numbered pixels to help select the proper sub-image. If the imshow(...) function does not do this automatically, then use the following code to display the image, I:

```
RI = imref2d(size(I));
figure(1);
imshow(I, RI);
```

9.3.1 Low-Pass Spatial Filtering Using a Moving Average Filter

A simple 3×3 or $N = 3$ *moving-average* filter has the form:

$$\mathbf{H}_3 = \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

Write out the two-dimensional filter, $\mathbf{H}_5$, for a moving-average filter with size $N = 5$. For this filter style (a subset of low-pass filters), the sum across the matrix elements should equal 1.

In MATLAB, create three moving-average filters with dimensions $N = \{5, 9, 13\}$ and use them to filter the image `I`. What is the effect of increasing the filter size in parts of the image with small details?

9.3.2 High-pass Spatial Filtering

The image filters 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 passes through the mask's middle row. 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.

For the image `I`, filter the image using the four following filters:

$$\mathbf{H}_a = \begin{bmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{bmatrix}, \quad \mathbf{H}_b = \begin{bmatrix} -1 & -1 & 2 \\ -1 & 2 & -1 \\ 2 & -1 & -1 \end{bmatrix}, \quad \mathbf{H}_c = \begin{bmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{bmatrix}, \quad \mathbf{H}_d = \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix}. \tag{9.1}$$

Describe your results for each case. Which edges were highlighted in each case? The MATLAB commands for the first case are given here:

```
1 H_a = [-1,-1,-1; 2,2,2; -1,-1,-1];
```

```
I_a = imfilter(I, H_a);
```

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

9.3.3 Filtering Noise

Noise often has a high spatial frequency; this appears as 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 much from the image. Speckle or impulsive noise is a very extreme type of noise that looks like white or black spots on an image. This section looks at how averaging filters affect the image.

Filtering Noise with Moving-Average Filters

Display the images `I_g` (the 'g' stands for Gaussian noise) and `I_sp` (the 'sp' stands for salt and pepper noise) in gray-scale. Two different types of noise have corrupted these images. You will use an averaging filter to remove the noise in this image. Filter each image with 3×3 and 7×7 moving-average filters. 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?

The Effects of High-pass Filters

Using the images `I_g` and `I_sp`, filter each image with the high-pass filters, $\mathbf{H}_a$ and $\mathbf{H}_c$ from Eq. (9.1). Display the results in image pairs.

Discuss the results: focus on how clear the edges in each direction appear. Does the filtering work better for one type of noise? If so, why?

9.3.4 Isolating Edges from Noisy Images

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 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.[3] 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 then filters and thins the edges. In this section, we implement the first two steps.

Smoothing

Apply a low-pass filtered 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 mask,[4][5], which smooths the edges of the filter. Use the filter parameters below to filter the image `I_g`:

```
H_gauss = [
    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;
```

Store the result of this filtering in the variable `I_g_smoothed`. Display `I_g_smoothed` and comment on what you see in the image.

Edge Detection

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,[4][5] $\mathbf{H}_x$ and $\mathbf{H}_y$, on `I_g_smoothed` (the result from Sec. 9.3.4) to produce two separate results, called $\mathbf{G}_x$ and $\mathbf{G}_y$, which contain images filtered in the x and y directions,

respectively. The contents of the Sobel edge detection filters are defined as:

$$\mathbf{H}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \qquad \mathbf{H}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

Intensity Gradient

Finally, combine the two filtered images, $\mathbf{G}_x$ and $\mathbf{G}_y$, from the preceding section, to compute the intensity gradient:[4][5]

$$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}.$$

Before utilizing either `G_x` or `G_y` to implement the equation above, ensure that they are of the proper type: double floating-point numbers. If they are not of the proper type, *typecast* them to double floating-point numbers using the `double(...)` MATLAB built-in function. Remember, in MATLAB, element-wise exponentiation is done using the dot caret (`.^`) operator.

Display the intensity gradient image, $\mathbf{G}$, using the command `imshow(uint8(G))` and comment on your results from Sec. 9.3.3 above. Are the edges clearer in one case or the other?

Thought Question

What will happen if the order of processing for a linear, spatially-invariant (LSI) low-pass filter, then an LSI high-pass filter is reversed for edge detection? Explain if order matters or not.

9.4 Report Checklist

Work with lab partner(s) to create a lab report for this lab exercise. Be sure to include the names of each person in your group on the report. This report should be a typed document that outlines the basic procedure of the lab exercise and should include any results you were asked to obtain. Your report should follow a *cohesive narrative*—it should always be apparent *why* you have presented the results you have. Do not simply insert results without (1) motivating the

problem and (2) providing an analysis of the data. Ensure that your report is grammatically correct with complete sentences. This does not mean your report must be wordy with filler—in fact, avoid filler and state your ideas concisely.

Be sure the following are included in your report:

- Section 9.2.2: Describe which region of Fig. 9.2 has the *lowest* spatial frequency.
- Section 9.2.3: Compute the horizontal spatial *period* of the image in Fig. 9.3 in terms of pixels/cycle.
- Section 9.2.5: Manually compute the value of the pixel at (3, 3) for the filter mask and image matrix provided in Fig. 9.4.
- Section 9.3.1: Write the contents of an $N = 5$ moving-average filter, $\mathbf{H}_5$.
- Section 9.3.1: Create three moving average filters of sizes $N = \{5, 9, 13\}$ in MATLAB and apply them to the image: `I`. Describe the effect of increasing the filter size.
- Section 9.3.2: In MATLAB, filter the image, `I`, using the four filters given in Eq. (9.1). Provide plots of each of the results. Describe the effects for each case; explain which types of edges were accent/highlighted.
- Section 9.3.3: Filter the `I_g` and `I_sp` images using 3×3 and 7×7 moving-average filters. Include a figure depicting each of the results where each figure contains before-and-after shots of the image. Include a discussion comparing the results for filtering different types of noise.
- Section 9.3.3: Filter the `I_g` and `I_sp` images using $\mathbf{H}_a$ and $\mathbf{H}_c$ from Eq. (9.1). Include a figure depicting each of the results where each figure contains before-and-after shots of the image. Include a discussion comparing the results for filtering different types of noise; focus on how clear the edges in each direction appear.
- Section 9.3.4: Filter `I_g` using the definition of `H_gauss` provided and store the result in the variable `I_g_smoothed`. Include the before-and-after images and comment on the results.
- Section 9.3.4: Include a plot of the final intensity gradient image, $\mathbf{G}$, and comment on the results. Compare the results to those produced using Sobel Edge detection in Sec. 9.3.3.
- Section 9.3.4: Consider what would happen if the order of processing for a linear, spatially-invariant (LSI) low-pass filter, then an LSI high-pass filter was reversed. Would this matter in terms of the result?

9.R References

[1] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 3rd ed. Addison–Wesley, 1992, ISBN: 0201508036.

[2] NASA, *Pathfinder on mars*, 1997. [Online]. Available: https://mars.nasa.gov/resources/8642/pathfinder-on-mars/.

[3] J. Canny, "A computational approach to edge detection," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–98, 1986. DOI: 10.1109/TPAMI.1986.4767851.

[4] B. Green. "Canny edge detection tutorial," Drexel University. (2002), [Online]. Available: https://web.archive.org/web/20150207095004/http://dasl.mem.drexel.edu/alumni/bGreen/www.pages.drexel.edu/_weg22/can_tut.html.

[5] P. K. Kalra. "Canny edge detection," Indian Institute of Technology Delhi. (2009), [Online]. Available: https://web.archive.org/web/20160115112215/http://www.cse.iitd.ernet.in/~pkalra/csl783/canny.pdf.

Pulse-width Modulation and Filter Design

PWM Encoding and Decoding

Overview

In this lab exercise, students will explore pulse-width modulation (PWM). Students will simulate a PWM encoder/decoder using the MATLAB script `pwm_sim.m` (available in this exercise's *supplementary materials* or in Appendix 10.B). Students will breadboard a comparator circuit using components from their Lab Kit and an LM293 op-amp provided by the Electronics and Technology Group. Students will use their comparator circuit along with the CyDAQ and DAD to implement a PWM encoder/decoder and verify its operation by streaming audio from their PC through the circuit to an external speaker.

Learning Objectives

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

1. Relate the data points in a message signal to the widths of pulses within a pulse-width modulation scheme.
2. Compute the pulse width that properly encodes a given data point within a pulse-width modulation scheme.
3. Relate the process of decoding pulse-width modulation to averaging with a low-pass filter.
4. Compute the cutoff frequency required to successfully decode a band-limited message signal encoded using a pulse-modulation scheme.
5. Implement and verify the operation of a pulse-width modulation encoder using a comparator circuit.

Materials

- (1×) DAD
- (1×) CyDAQ
- (1×) Hardware Speaker
- (1×) Lab Kit (with *Assorted Wires* and *Breadboard*)
- (1×) Jumper Splitter
- (2×) 3.5 mm Male-to-Breakout Aux Cables
- (8×) Female-to-Male Jumper Cables
- (1×) LM293 OpAmp
- (1×) 0.47 µF Capacitor
- (1×) 1 kΩ Resistor (Brown–Black–Red)
- (2×) 100 kΩ Resistor (Brown–Black–Yellow)

10.1 Introduction

In this lab exercise, you will explore pulse width modulation (PWM). First, you will implement and confirm the proper operation of a simple PWM scheme using the CyDAQ and DAD. Next, you will simulate a more complex PWM scheme in MATLAB. Finally, you will build a comparator circuit using components from your Lab Kit and the Electronic and Technology group. You will use this circuit in conjunction with the DAD, CyDAQ and external speaker to implement a hardware PWM encoder/decoder.

10.2 Pulse-width Modulation

Pulse-width modulation (PWM) is an encoding scheme that encodes a bounded message signal by modulating the pulse widths of a carrier signal. PWM can be thought of as a means of both *sampling* a bounded message signal, $x(t)$, to a set of data points, $x[n]$, and *digitizing* it by restricting its domain to logic levels.

The exact workings of PWM are best explained through example. Let $x(t)$ be a message signal bounded between 0 and 5. Let's consider a simple case where $x(t)$ is a constant value $x(t) = 2$.

To encode $x(t)$ as a PWM signal, we need to know two things: (1) the bounds of the message signal and (2) the pulse period, T_0. Upon first thought, it might seem reasonable to assume that the bounds of the message signal is simply 2. However, we chose $x(t) = 2$ as a contrived example; the true bounds of the signal were stipulated to lay between 0 and 5. We are free to choose our own pulse period for this example. Let's choose $T_0 = 1\,\text{ms}$. Sampling $x(t)$ with a sampling period of T_0 gives $x[n] = 2$.

To encode $x[n]$ as a PWM signal, we need to compute the pulse widths for each $x[n]$. The width of the nth pulse, denoted $T_{\text{on}}[n]$, is defined as $T_{\text{on}}[n] = D[n] \cdot T_0$, where $D[n]$ is the *duty cycle* of the nth pulse. The equation for computing the duty cycles is:

$$D[n] = x[n]/V_{\text{max}},$$

where V_{max} is the upper bound of the message signal, $x(t)$.

For our example, $V_{\text{max}} = 5$. Consequentially, $D[n] = 2/5 = 0.4$ and $T_{\text{on}} = 0.4 \times 1\,\text{ms} = 400\,\mu\text{s}$. Now that we know the pulse widths, T_{on}, that encode $x[n]$, we can construct our PWM signal. At the beginning of each pulse period, the pulse rises to a height of V_{max} and remains there for a duration of T_{on} after which it falls to V_{min} stays there for the remainder of the pulse period $T_{\text{off}} = T_0 - T_{\text{on}}$. The PWM representation of our example is illustrated in Fig. 10.1.

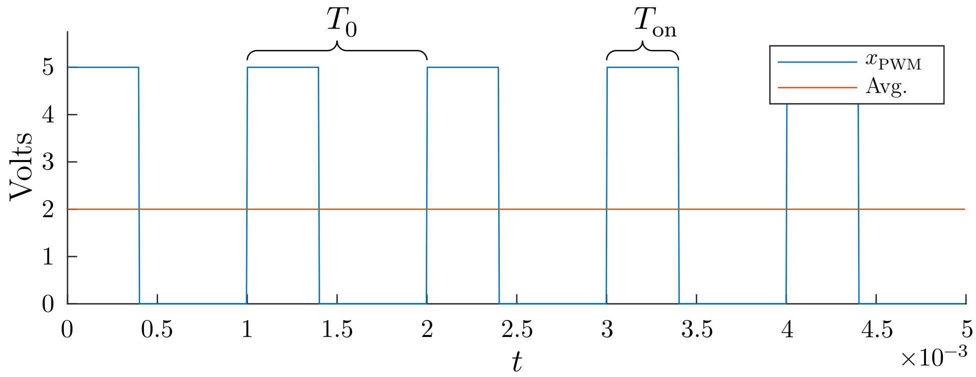


Figure 10.1: A PWM signal encoding the message signal $x(t) = 2$, with a upper bound $V_{\text{max}} = 5$ and a pulse period $T_0 = 1\,\text{ms}$. The signal AVG plots the average value of the PWM signal during each pulse period T_0.

The graph in Fig. 10.1 also depicts a signal, 'AVG', that plots the average value of the PWM signal for each pulse period T_0. Notice that this average value tracks with our underlying message signal. This property suggests a simple mechanism for decoding PWM signals.

10.3 Decoding Pulse-width Modulation

Previous lab exercises have covered the topic of filtering and frequency response, but it is worth reviewing some of the fundamentals before moving forward. Fig. 10.2 depicts the frequency response of a first-order, low-pass filter with a cutoff frequency: $f_c = 2\,\text{kHz}$. The cutoff (or 'mid corner') frequency is an important characteristic of the filter as it serves as the threshold before which frequencies are passed and after which frequencies are attenuated. From previous lab exercises, we know that the CyDAQ can implement this type of filter. However, first-order filters of this class can be easily realized using an RC circuit, such as the one depicted in Fig. 10.3. This ease of implementation makes first-order, low-pass filters a fundamental building block in more complex circuits such as those implementing encoding schemes.

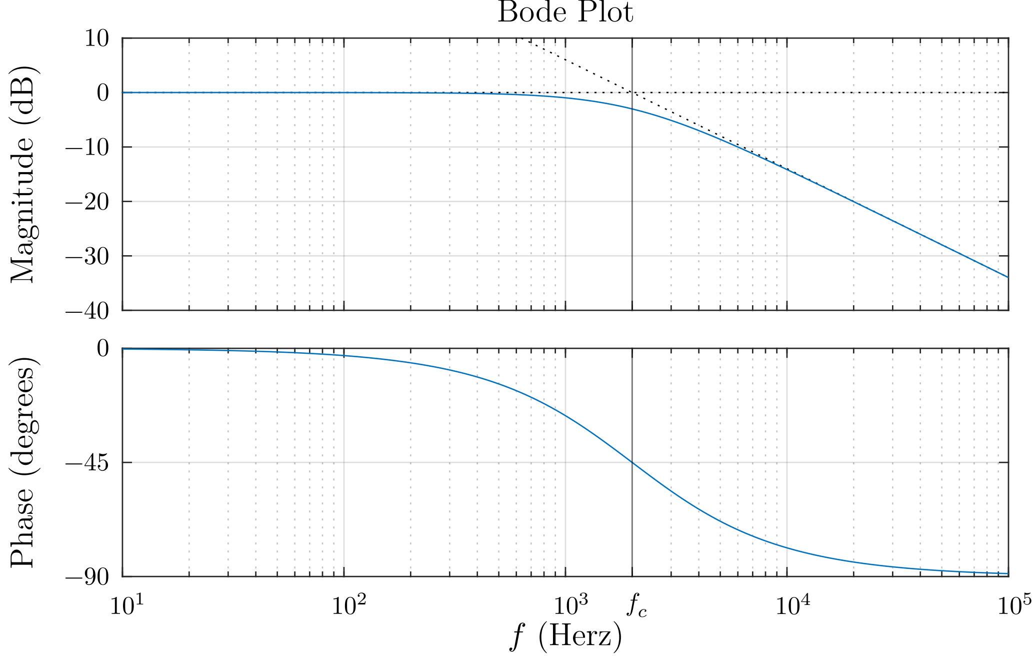


Figure 10.2: A first-order, low-pass filter with cutoff frequency $f_c = 2\,\text{kHz}$.

A low-pass filter behaves similarly to a moving average. In your lab report, provide some

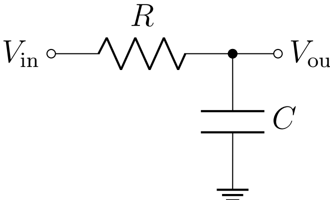


Figure 10.3: A first-order, low-pass RC filter circuit.

intuition that explains this behavior. As was alluded to in the previous section, the average value of the nth pulse period of a PWM signal corresponds to the discretized message signal, $x[n]$. Consequentially, a low-pass filter can be applied to a PWM to obtain a relatively close approximation of the message $x(t)$. All that is required is that the cutoff frequency of the low-pass filter is much less than the pulse *period* $1/T_0$. In other words, $f_c << T_0$.

The `WaveForms Software` is capable of generating periodic pulses with a custom duty cycle, and past lab exercises have used the CyDAQ as a customizable filter. Consequentially, it is possible to use the DAD to simulate a PWM system encoding a custom message and use the CyDAQ as a decoder. Let's suppose we want to encode the constant message $x(t) = 1.25\,\text{V}$ using a PWM encoding scheme in which $V_{\text{max}} = 3.3\,\text{V}$ and $T_0 = 10\,\mu\text{s}$. Compute the duty cycle, D, needed to encode $x(t)$ and record your result and calculations in your lab report.

With these calculations completed, let's configure the DAD and CyDAQ for this PWM experiment.

- Connect the **DAD's Ground Terminal** (`GND`) and **Scope Ch. 1 Negative Terminal** (`CH1-`) to the **CyDAQ's Negative V_{in} Terminal** (`Vin-`) using the **Jumper Splitter**.
- Connect the **DAD's WaveGen Terminal** (`W1`) and **Scope Ch. 1 Positive Terminal** (`CH1+`) to the **CyDAQ's Positive V_{in} Terminal** (`Vin+`) using the **Jumper Splitter**.
- Connect the **DAD's Scope Ch. 2 Negative Terminal** (`CH2-`) to the **CyDAQS Negative V_{out} Terminal** (`Vout-`).
- Connect the **DAD's Scope Ch. 2 Positive Terminal** (`CH2+`) to the **CyDAQS Positive V_{out} Terminal** (`Vout+`).

The pin-outs for the CyDAQ and DAD are given in Appexendix 10.A. Launch the `CyDAQ Software` and send the following configuration:

- **Sampling Rate**: 48000

- **Input**: Analog In
- **Filter**: 1st Order Low Pass
- **Mid Corner**: 2000

Launch the `WaveForms Software` and open the **Wavegen**. Enter the following configuration:

- **Type**: Pulse
- **Frequency**: 100 kHz
- **Amplitude**: 3.3 V
- **Offset**: 0 V
- **Symmetry**: Duty Cycle
- **Phase**: 0°

In the **Symmetry** field, enter the duty cycle (as a percentage) you computed.

Launch a **Scope** tab, and compare the input PWM signal to the filtered message. Does this look how you would expect it to? Include an image of the scope capture in your lab report and discuss why the filtered message signal looks the way it does.

10.4 Encoding Pulse-width Modulation

Until now, we've constructed simple PWM encodings based on a constant message signal. However, PWM is more than capable of encoding non-constant (but still bounded) messages. Consider a sinusoidal message signal $x(t) = \cos(2\pi f_a t)$ for some frequency f_a. Selecting a sufficiently small $T_0 << 1/f_a$ will result in a PWM system that can encode $x(t)$ with relative accuracy.

As mentioned in the previous section, each nth pulse period encodes a single value "sampled" value, $x[n]$, of the message. The word *sampled* is in quotes because the actual implementation of the PWM encoder may not be sampling $x(t)$ in a traditional sense. It is certainly possible to implement a PWM encoder on an embedded system that samples a continuous message signal, $x(t)$, at a constant rate and computes the required pulse width for each pulse period. There is, however, another common method PWM encoding method that introduces less overhead: a comparator.

Consider an op-amp configured as a comparator. The output saturates high when the positive terminal has a greater voltage than the negative terminal, else the voltage saturates low. Suppose

we feed a message signal, $x(t)$, into the positive terminal, V_{in}, of a comparator. If we generate a sawtooth signal, V_{saw}, that ranges from 0 to V_{max} with a period of T_0 and feed this into the negative terminal of a comparator, the output of the comparator will be the PWM encoding of the message signal $x(t)$. This configuration is depicted in Fig. 10.4. While not sampling in the traditional sense (the spacing between samples may not be constant depending on when the sawtooth overtakes the message signal), the PWM encoding produced remains relatively accurate.

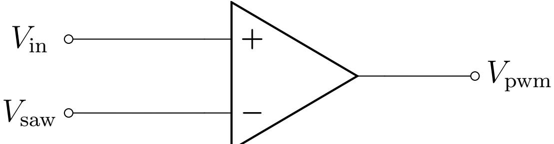


Figure 10.4: Diagram of Analog PWM Encoder.

10.4.1 MATLAB Simulation

We will now simulate an analog PWM encoder with a sinusoidal message signal in MATLAB. The skeleton of a MATLAB script, `pwm_sim.m` has been provided to you in this lab exercise's *supplementary materials* as well as in Appendix 10.B. Create a new folder on your PC to serve as your project directory and place `pwm_sim.m` within it. Launch MATLAB and open `pwm_sim.m`. You'll notice that the last half of the script has been commented out. This was done to allow you to run the script to confirm that you have completed the first half of the script correctly before moving on to the next half.

Your task is to fill in each of the `TODO` portions of the code. The first `TODO` statement is within the sawtooth function in the definition of `x_saw`. Read the comments above this definition to understand the requirements for the `x_saw` variable. You must replace the `TODO` statement with an expression that satisfies the requirements in the comment. This expression may require that you use other variables defined earlier in the script, so be sure to investigate the script thoroughly. Remember that you can use the `help` command to get help for functions you aren't familiar with.

The second `TODO` statement is within the definition of the message signal, `vin`. Again, the requirements that `vin` should satisfy are specified in a comment above the definition.

Once you've replaced the `TODO` statements with the proper expressions, run the script to verify the results. You should see a graph with three overlaid signals: the message signal V_{in}, the sawtooth signal, and the PWM output. If you have correctly filled out the script, each of these signals will be bounded between 0 V and 5 V, and the message signal should span exactly three periods. Include this copy of this plot in your lab report. Discuss the meaning of each signal in the plot and explain how they relate to one another.

Once you've completed the first half of the script, delete line 63 to uncomment the second half. This portion of the script deals with *decoding* the PWM output and comparing the result to the original input. Recall from Sec. 10.3 that PWM signals are decoded by filtering the output. Each of the `TODO` statements in this portion of the script are for configuring this filter as parameters to the `butter(...)` function. This function creates a *Butterworth filter*, a special class of filters with maximally flat pass and stop bands. Use the `help` command on the `butter(...)` function to learn the format of the parameters that it expects. You will have to select a cutoff frequency for this filter. It will need to be low enough to be well below pulse frequency but high enough to avoid attenuating the content of the message signal.

Once you configured the filter, run the script. You should see two more figures appear, one showing the Bode of the filter and another showing the original message V_{in} overlaid with the filtered PWM out. If done correctly, the filtered PWM output should reach a steady state that is a slightly delayed version of V_{in}. The peaks and troughs of the filtered PWM signal should span the same range as the peaks and troughs of V_{in}. If the peaks/troughs do not span the proper range, then the message signal has been attenuated by the filter—indicating an incorrect choice of filter cutoff frequency. Similarly, if high-frequency content not present in V_{in} is visible in the filtered PWM output, the filter cutoff frequency has also been chosen incorrectly. Once you've confirmed that the script has been properly completed, include a copy of the V_{in}/filtered PWM plot in your lab report. Discuss your choice of cutoff frequency. Why is the filtered PWM signal delayed from V_{in}? What is the phase shift (in degrees) from V_{in} to the filtered output? *Hint*: the Bode plot of the filter may be useful here.

10.4.2 PWM Circuit

Now that you've completed a simulation of a PWM encoder, we will proceed to build the real thing by breadboarding a comparator circuit, that takes V_{in} and V_{saw} as inputs and produces the PWM signal V_{pwm}. Your PC's headphone port will provide the message signal (V_{in}), the DAD will provide the sawtooth function (V_{saw}) and oscilloscope, and the CyDAQ will provide the filter for decoding the PWM signal V_{pwm}. Use the components in your Lab Kit, along with the LM293 op-amp provided in the lab to construct the circuit in Fig. 10.5. You must use the LM293 op-amp, other op-amps in your Lab Kit *cannot* be used as substitutes. The pinout for the LM293 op-amp is given in Fig. 10.6.

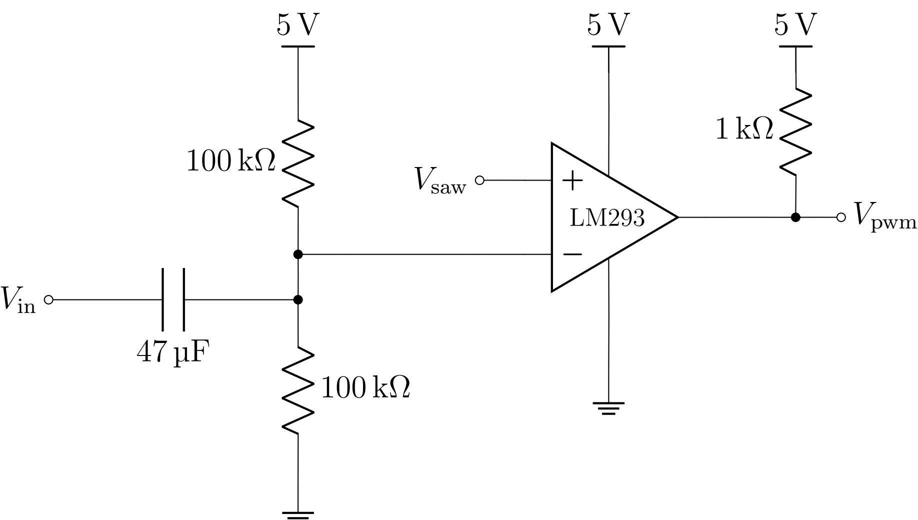


Figure 10.5: Circuit For PWM Encoder.

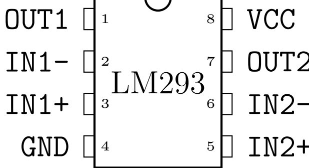


Figure 10.6: LM293 Pinout

Once you have built the circuit depicted in Fig. 10.5 you will need to connect it to your PC headphone port, the CyDAQ's V_{in} and V_{out} ports, as well as the DAD's WaveGen and Scope. You will need to make the following connections:

- Connect the **CyDAQ's 5 V Sensor Port** (`J11`) to the **Breadboard's Power Rails** using **Female-to-Male Jumper Cables**.
  - Connect the **CyDAQ's 5 V Sensor Ground Terminal** (*leftmost* terminal of `J11`) to the **Breadboard's Ground** (Negative Rail).
  - Connect the **CyDAQ's 5 V Sensor 5 V Terminal** (*middle* terminal of `J11`) to the **Breadboard's** V_{ss} (Positive Rail).
- Connect the **PC's Headphone Terminal** to the **Breadboard's PWM Circuit** using a **3.5 mm Aux Breakout Cable**.
  - Connect the **PC Headphone Ground** (**Black Breakout Wire**) to the **Breadboard's Ground** (Negative Rail).
  - Connect the **PC Headphone Right Channel** (**Red Breakout Wire**) to the **PWM Circuit's** V_{in} (see Fig. 10.5).
  - The **PC Headphone Left Channel** (**White Breakout Wire**) is *unused*.
- Connect the **DAD's WaveGen Ch. 1** (Sawtooth) to the **Breadboard's PWM Circuit**.
  - Conect the **DAD's Ground Terminal** (`GND`) to the **Breadboard's Ground** (Negative Rail).
  - Connect the **DAD's Waveform Generator 1** (`W1`) to the **PWM Circuit's** V_{saw} (see Fig. 10.5).
- Connect the **PWM Circuit's** V_{pwm} to the **CyDAQ's** V_{in} **Terminals** using **Female-to-Male Jumper Cables**.
  - Connect the **Breadboard's Ground** (Negative Rail) to the **CyDAQ's** V_{in} **Negative Terminal** (`Vin-`).
  - Connect the **PWM Circuit's** V_{pwm} (see Fig. 10.5) to the **CyDAQ's** V_{in} **Positive Terminal** (`Vin+`).
- Connect the **CyDAQ's** V_{out} **Terminal** to the **Breadboard** using **Female-to-Male Jumper Cables**. If a single cable is too short, string together two cables to create longer **Female-to-Male cables**.
  - Connect the **CyDAQ's** V_{out} **Negative Terminal** (`Vout-`) to an **Unused Row on the Breadboard**.
  - Connect the **CyDAQ's** V_{out} **Positive Terminal** (`Vout+`) to an **Unused Row on**

the Breadboard.

- Connect the **CyDAQ's** V_{out} (via Breadboard) to the **DAD's Scope Ch. 2**.
  - Connect the **CyDAQ's** V_{out} **Negative Terminal** (`Vout-`) to the **DAD's Scope Ch. 2 Negative Terminal** (`CH2-`).
  - Connect the **CyDAQ's** V_{out} **Positive Terminal** (`Vout+`) to the **DAD's Scope Ch. 2 Positive Terminal** (`CH2+`).
- Connect the **Speaker's Power Connector** to the **Breadboard's Power Rails**.
  - Connect the **Breadboard's Ground** (Negative Rail) to the **Speaker's Power Negative Terminal** (**Black Wire**).
  - Connect the **Breadboard's** V_{ss} (5 V) to the **Speaker's Power Positive Terminal** (**White** or **Red Wire**).
- Connect the **CyDAQ's** V_{out} (via Breadboard) to the **Speaker's Audio-In** using a **3.5 mm Breakout Cable**.
  - Connect the **CyDAQ's** V_{out} **Negative Terminal** (`Vout-`) to the **Speaker's Audio Ground** (**Black Breakout Wire**).
  - Connect the **CyDAQ's** V_{out} **Positive Terminal** (`Vout+`) to the **Speaker's Audio Right Channel** (**Red Breakout Wire**).
  - The **Speaker's Audio Left Channel** (**White Breakout Wire**) is *unused*.
- Connect the **DAD's Scope Ch. 1** to the **Breadboard** (for probing).
  - Connect the **DAD's Scope Ch. 1 Negative Terminal** (`CH1-`) to the **Breadboard's Ground**.
  - Connect the **DAD Scope Ch. 1 Positive Terminal** (`CH1+`) to the **Probe Location** (whichever part of the circuit you would like to verify).

The pin-outs for the CyDAQ and DAD are given in Appexendix 10.A. Open the **Wavegen** in the `WaveForms Software` and enter the following configuration:

- **Type**: RampUp
- **Frequency**: 200 kHz
- **Amplitude**: 2.5 V
- **Offset**: 2.5 V
- **Symmetry**: 100%

- **Phase**: 0°

This will generate a sawtooth wave from 0 V to 5 V. Open the `CyDAQ Software` and send the following configuration:

- **Sampling Rate**: 48000
- **Input**: Analog In
- **Filter**: 6th Order Low Pass
- **Mid Corner**: Cutoff

In the **Mid Corner** field, enter the cutoff you chose in Sec. 10.4.1.

Begin streaming audio from your PC into the circuit. Choose an audio source that is long in duration, and has a high volume without being distorted. Classical organ or brass pieces are good candidates to stream. If your circuit was constructed correctly, you should hear your streamed audio through the external speaker. Using the DAD's Scope, probe the output of CyDAQ V_{in} on Scope Ch. 1 and the CyDAQ's V_{out} on Scope Ch. 2. Confirm that Scope Ch. 1 depicts a PWM signal while Scope Ch. 2 depicts the decoded audio signal. Take a snapshot of the WaveForms scope displaying both the PWM and decoded audio waveforms. Include this image in your lab report and discuss the quality of the decoded audio.

10.5 Report Checklist

Work with lab partner(s) to create a lab report for this lab exercise. Be sure to include the names of each person in your group on the report. This report should be a typed document that outlines the basic procedure of the lab exercise and should include any results you were asked to obtain. Your report should follow a *cohesive narrative*—it should always be apparent *why* you have presented the results you have. Do not simply insert results without (1) motivating the problem and (2) providing an analysis of the data. Ensure that your report is grammatically correct with complete sentences. This does not mean your report must be wordy with filler—in fact, avoid filler and state your ideas concisely.

Be sure the following are included in your report:

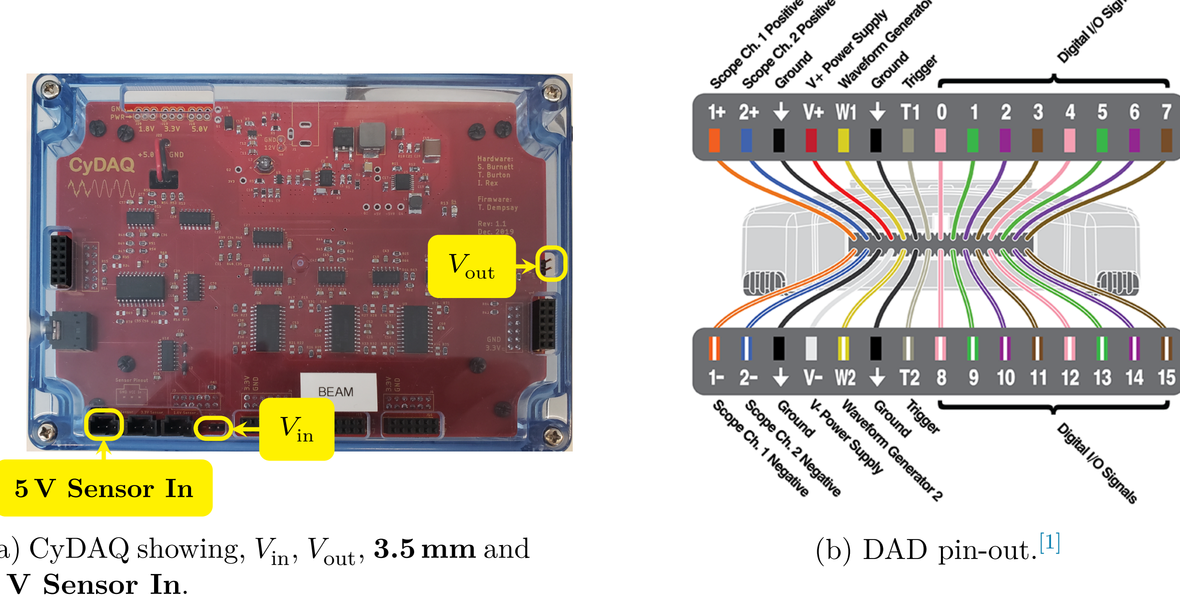
- Section 10.3: Explain why a low-pass filter behaves as an averaging function.
- Section 10.3: Include your duty cycle calculations in your lab report.

- Section 10.3: Include a scope capture of the PWM and filtered message signals. Explain why the filtered message signal appears the way that it does.
- Section 10.4.1: Include a copy of the V_{in}/sawtooth/PWM plot produced by the MATLAB simulation. Discuss the meaning of each of the signals in this plot and explain how they relate to one another.
- Section 10.4.1: Include a copy of the V_{in}/filtered PWM output plot in your report. Discuss your choice of cutoff frequency. Why is the filtered PWM signal delayed from V_{in}? What is the phase shift (in degrees) from V_{in} to the filtered output?
- Section 10.4.2: Include an image of the WAveforms scope displaying both the PWM signal and decoded audio waveforms. Discuss the audio quality of the decoded PWM signal.

10.6 Acknowledgments

This lab exercise is based on an exercise drafted by **Isaac Rex**.

10.A Hardware Ports



(a) CyDAQ showing, V_{in}, V_{out}, **3.5 mm** and **5 V Sensor In**.

(b) DAD pin-out.[1]

Figure 10.7: Pin-outs/terminals for CyDAQ and DAD.

Figure 10.8: CyDAQ Speaker showing **3.5 mm Audio** terminal and **Power Connector**.

10.B PWM Simulation MATLAB Script

Listing 10.1: `pwm_sim.m`

```
close all; clearvars;
% PWM Simulation
% Written By: Isaac Rex, 2020
% Updated By: Aaron Fonseca, 2024

% **Fill-in all TODOs**

A_saw = 5;      % 5V amplitude
A_sin = 2;      % Amplitude of vin
f_saw = 200e3;  % Sawtooth frequency. This is the effective sampling rate of
                % the input signal and will be the same as the PWM frequency
f_sin = 20e3;   % Sine wave frequency (Hz)
N_periods_sin = 3; % Number of sine wave periods to simulate

%% Generate Sawtooth Wave
% How many periods (or samples) to simulate. Simulate enough periods of
% the sawtooth to capture three full periods of vin
N_saw_cycles = ceil(f_saw/f_sin) * N_periods_sin;

dt_sim = 1/(f_saw * 100); % This is our simulations time-step. That is, how
```

```
21                          % fine of resolution are we simulating at.
22
23  t = 0:dt_sim:N_saw_cycles/f_saw-dt_sim; % Simulate time vector
24
25  % *********************************************************************
26  %  TODO: Fill in the sawtooth() argument, use: `help sawtooth` if needed
27  % *********************************************************************
28  % Generates a sawtooth wave that goes from 0 to A_saw at f_saw frequency
29  x_saw = (sawtooth( TODO ) + 1) * A_saw/2;
30
31  %% Generate vin
32
33  % *********************************************************************
34  %  TODO: Fill in the statement. vin should be a sin wave with frequency
35  %  f_sin Hz, centered at A_saw/2 with amplitude A_sin (e.g., for A_sin = 2
36  %  and A_saw = 5, the sine wave should range between 0.5 and 4.5)
37  % *********************************************************************
38  % Generate and center input (usually input is centered around the sawtooth)
39  vin = TODO;
40
41  %% Simulate the comparator
42  pwm = A_saw * (x_saw < vin);
43
44  %% Plot waveforms
45  figure(1)
46  clf;
47
48  hold on;
49  plot(t, pwm, 'LineWidth', 2);
50  plot(t, vin, 'LineWidth', 2);
51  plot(t, x_saw, 'LineWidth', 2);
52  grid on;
53
54  wf_axis = gca;                % Get figure handle for editting plot
55  set(wf_axis, 'fontsize', 22)  % Set figure's fontsize
56  set(wf_axis.XAxis, 'Exponent', -6) % Set xlabel exponent to -6 (us)
```

```
57  xlabel('Time (sec)', 'FontSize', 22)
58  ylabel('Volts', 'FontSize', 22)
59  legend('PWM Output', 'V_{in}', 'Sawtooth', 'fontsize', 20)
60  title('V_{in} Input And PWM Output', 'FontSize', 24)
61
62  % Delete the FOLLOWING LINE (LINE 63) to uncomment the following section
63  %{
64  % **************************************************************************
65  %  Put in the filter parameters below. You need to choose a cutoff
66  %  frequency and filter type. Type 'help butter' for more information
67  % **************************************************************************
68  %% Design output filter
69  filt_order = 6;
70  filt_cutoff_hz = TODO;
71  filt_type = TODO;
72
73  % Get filter's TF coefficents
74  [b, a] = butter(filt_order, 2*pi*filt_cutoff_hz, filt_type, 's');
75  H = tf(b, a); % Generate TF
76
77  %% Make Bode plot of filter
78  % bodeoptions isn't strictly necessary, but is really nice when making
79  % plots for reports (or if you just want frequency units in Hz)
80  P = bodeoptions;
81  P.FreqUnits = 'Hz';
82  P.Grid = 'on';
83
84  figure(2)
85  bode(H, P)
86
87  %% Simulate the PWM signal going through the filter
88  vout = lsim(H, pwm, t); % Filter the PWM signal
89
90  %% Plot the output
91  figure(3)
92  clf;
```

```
93 plot(t, vin, 'LineWidth', 2);
94 hold on;
95 plot(t, vout, 'LineWidth', 2);
96 grid on
97
98 set(gca, 'fontsize', 22)
99 xlabel('Time (sec)', 'FontSize', 22)
100 ylabel('Volts', 'FontSize', 22)
101 legend('V_{in}', 'Filtered Output', 'fontsize', 20)
102 title('V_{in} Input And Filtered Output', 'FontSize', 24)
103 %}
```

10.R References

[1] "Analog discovery 2 reference manual," Digilent. (2018), [Online]. Available: http://digilent.com/reference/test-and-measurement/analog-discovery-2/reference-manual.

Sampling and Aliasing

An Exploration of the Sampling Theorem

Overview

In this lab exercise, students will explore sampling and its implications in the frequency domain. Students will plot the frequency content of various sampled signals using the `plot_spectrum.m` script (available in this exercise's *supplementary materials* or in Appendix 11.A). Students will use the CyDAQ to capture sampled sinusoids at various sampling rates.

Learning Objectives

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

1. Relate the number of samples in a finite length discrete time signal to the number of frequency bins in its Discrete Fourier Transform.
2. Plot the frequency spectrum of sampled signals in MATLAB.
3. Infer the apparent frequency of aliased sinusoids.

Materials

(1×) DAD
(1×) CyDAQ

11.1 Introduction

In this lab exercise, you will explore the basic concepts of the sampling theorem and some of its exploits. First, you'll sample a signal near the Nyquist frequency and then reconstruct it in MATLAB, then you'll see the effects of undersampling a signal in the frequency domain using the FFT function both on the DAD and in MATLAB.

11.2 Background

11.2.1 Sampling in the Frequency Domain

Sampling a continuous-time signal, $x(t)$, at a rate of f_s can be thought of as multiplying $x(t)$ with a delta train whose spacing between deltas is $T_s = 1/f_s$. Recall that multiplication in the time domain is convolution in the frequency domain and the Fourier transform of a delta train is a stretched and scaled delta train. Consequentially, sampling at a rate of f_s in the time domain results in scaled, f_s periodic replications in the frequency domain.

Recall also that signals that are real in the time domain are conjugate symmetric in the frequency domain. This means that a signal with a bandwidth, B, has a frequency content between $-B$ and B hertz. When a signal with bandwidth, B, is sampled at a rate of f_s, what used to be negative frequency content between $-B$ and 0 hertz now also exists from $f_s - B$ to f_s (and vice versa across multiple of f_s etc) due to periodic replication. Thus, to ensure that no unintended interference occurs due to frequency overlap (or *aliasing*), one should choose the sampling rate f_s such that $f_s \geq 2B$. This is the origin of the Nyquist theorem.

A more concrete example of the effect of sampling is given in Fig. 11.1. Here, a 20 Hz sinusoid is sampled at a rate of 50 Hz. After sampling we can see that additional frequency content has appeared at ± 30 Hz ± 70 Hz, etc.

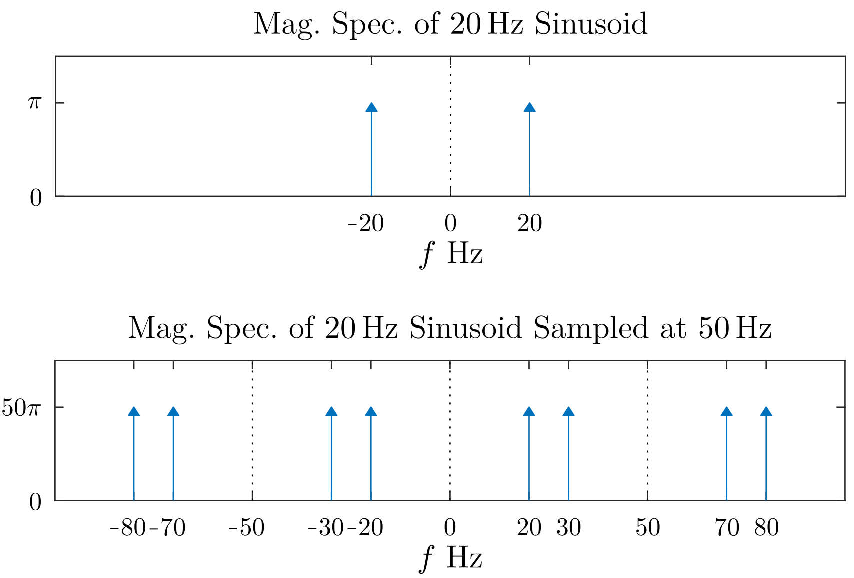


Figure 11.1: Original spectrum vs. sampled spectrum.

11.2.2 The Discrete Fourier Transform

Digital systems cannot represent continuous-time signals (this would require them to store not merely an infinite number of points, but an *uncountably infinite* number of points). Consequentially, digital systems do not (typically) utilize the Fourier transform, as this requires continuous data. Instead, digital systems employ the *Discrete Fourier Transform* (DFT). Usually, an efficient algorithm for computing the DFT called the *Fast Fourier Transform* (FFT) is used so the two terms are often used interchangeably.

If x is an N-length vector representing a discrete-time signal sampled at a rate of f_s, then the Discrete Fourier Transform of x, DFT$\{x\}$ is an N-length vector giving the sampled frequency spectrum between 0 and $\frac{N-1}{N} f_s$ hertz. As a consequence, we can gain insight into the effects of sampling at different rates by comparing the DFT spectrums.

11.3 Effects of Sampling

The MATLAB script, `plot_spectrum.m` has been provided to you in this lab exercise's *supplementary materials* as well as in Appendix 11.A. This script takes a vector, `x` containing a time domain signal, and a scalar `Fs` containing the sampling rate of `x`. Using all the knowledge of

MATLAB you've accumulated throughout the previous lab exercises, create a script in your project directory called `test_5_1k_at_20k.m` that does the following:

1. Generates $N = 5$ periods of a 1 kHz sine wave sampled at $f_s = 20$ kHz and stores it in the variable `x` with associated time vector `t`.
2. Generates a figure containing a time domain plot of the sine wave, `x`, with respect to the time vector, `t`. The axes of this plot should be properly labeled.
3. Generates a figure containing the DFT spectrum of `x` using the `plot_spectrum(...)` function.

In your lab report, include both the time domain and frequency spectrum plots produced by `test_5_1k_at_20k.m`.

Create a new script in your project directory called `test_50_1k_at_20k.m` and paste the contents of `test_5_1k_at_20k.m` into it. Modify your new `test_50_1k_at_20k.m` so that it generates $N = 50$ periods of a 1 kHz sine wave sampled at $f_s = 20$ kHz. Include both the time domain and frequency spectrum plots produced by `test_50_1k_at_20k.m` in your report. Compare the frequency spectrums produced by the `test_5_1k_at_20k.m` and `test_50_1k_at_20k.m` scripts. Discuss the effect(s) on the frequency spectrum of increasing the number of periods from $N = 5$ to $N = 50$.

Create a new script in your project directory called `test_50_8k_at_20k.m` and paste the contents of `test_50_1k_at_20k.m` into it. Modify your new `test_50_8k_at_20k.m` script so that is generates $N = 50$ periods of a 8 kHz sine wave sampled at $f_s = 20$ kHz. Include both the time domain and frequency spectrum plots produced by `test_50_8k_at_20k.m` in your report. Compare the frequency spectrums produced by the `test_50_1k_at_20k.m` and `test_50_8k_at_20k.m` scripts. Discuss the effect(s) on the frequency spectrum of increasing the frequency of the sine wave from 1 kHz to 8 kHz.

11.3.1 Sampling Near Nyquist

The Nyquist theorem states that the sampling rate, f_s, should be at least twice the maximum bandwidth of the signal to be sampled. However, while this mathematically guarantees that alias will not occur, having a sampling rate *close* to twice the maximum bandwidth presents

practical problems as well. In order to reconstruct the original spectrum from a sampled signal, the periodic replications must be removed via a lowpass filter. An ideal low pass filter would be a rect function in the frequency domain whose width from 0 Hz is f_s. Why might this be difficult to realize? In your lab report, discuss why ideal low-pass filters are difficult to realize and why frequency content close to the edge of the ideal rect function might be more prone to distortion if imperfect realizations of a low-pass filter are utilized.

We will now conduct experiments on the effects of sampling close to the Nyquist frequency. The CyDAQ has the ability to capture data at custom sampling rates and save the results to `.mat` files. We will use the DAD to generate high-fidelity sinusoids that will then be sampled by the CyDAQ. Make the following connections:

- Connect the **DAD's Waveform Generator 1** (`W1`) and the **DAD's Ground** (`GND`) **Jumper Wires** to the **CyDAQ's** V_{in} **Terminals**, (`Vin+`) and (`Vin-`), respectively.

Launch the `WaveForms Software` and open a new **Wavegen** tab. Enter the following configuration:

- **Type**: Sine
- **Frequency**: 1 kHz
- **Amplitude**: 1 V
- **Offset**: 0 V
- **Symmetry**: 50%
- **Phase**: 0°

Run the Wavegen. Launch the `CyDAQ Software` and send the following configuration:

- **Sampling Rate**: 2100 Hz
- **Input**: Analog In
- **Filter**: All Pass

Note that the **Sampling Rate** is a text-entry field in addition to a drop-down menu. You will need to enter 2100 Hz as a text entry since it is not in the drop-down menu.

Capture about one second of data using the **Start Sampling** button in the `CyDAQ Software`. Save the data as `dat_1000_at_2100.mat` in your project directory.

Create a new script in your project directory called `test_1000_at_2100.m` and paste the contents of `test_50_1k_at_20k.m` into it. Modify the script so that the contents of the data

signal, `x`, and time vector, `t` are loaded from `dat_1000_at_2100.mat`. Recall also that the `.mat` files generated by the CyDAQ application contain a variable called `data`, which is a matrix whose *first* column contains the time vector, `t`, and whose *second* column contains the signal amplitudes, `x`. Don't forget to update the sampling frequency, `Fs`, in your script to the proper value of `Fs = 2100`. Run the script and include both the time domain and frequency spectrum plots produced in your lab report. Does the time domain waveform look like a 1 kHz sine wave? You may need to zoom in to get a better view. Does the appearance of the time-domain signal comport with the content of the frequency spectrum? Discuss this in your lab report.

11.3.2 Upsampling for Reconstruction

We will now reconstruct a (20×) upscaled version of the sampled signal captured by the Cy-DAQ. Create a new script in your project directory called `upsample_1000_at_2100.m` and paste the contents of `test_1000_at_2100.m` into it. Before the plotting commands, create three new variables: `Fs_up`, `x_up`, and `t_up`.

Define `Fs_up` as twenty times `Fs`. Define `x_up` as `x_up = resample(x, P, Q, b)` where $P = 20$, $Q = 1$ and $b = 100$. This will upsample `x` by 20 times. The b parameter controls the "effectiveness" of the reconstruction filter where 100 indicates the reconstruction should be *very* effective. The upsampled signal, `x_up`, will have the following properties:

a. The upsampled signal, `x_up`, will have 20 times as many samples as the original signal, `x`.

b. The upsampled signal, `x_up`, will have a *new* sampling rate, `Fs_up`, which is 20 times the original sample rate `Fs`.

c. The upsampled signal, `x_up`, will be the same duration **in time (seconds)** as the original signal, `x`. This might not be intuitive at first (as `x_up` has 20 times as samples) but consider that the spacing (in time) *between* samples (which is related to the sampling rate) has changed as well.

Once you have generated the upsampled signal, `x_up`, you will need to generate a new time vector, `t_up`, that corresponds to your *new* sampling rate, `Fs_up`. This time vector, `t_up`, should have the *same* number of samples as `x_up`. You can find the number of samples in `x_up` using `N_up = length(x_up)` command in MATLAB. Your `t_up` vector must be a vector of `N_up` samples

beginning at 0 where the difference between consecutive samples is `1/Fs_up`.

Once you have defined `t_up` and `x_up` modify the plotting commands so that the time domain figure plots `x_up` with respect to `t_up` and the frequency spectrum is taken with respect to `x_up` and `Fs_up`. Verify that what you see makes sense. You may need to enlarge the view region of the time domain plot to get a proper view. Discuss whether the plots of the upscaled reconstruction comport with the plots produced by `test_1000_at_2100.m`.

11.3.3 Aliasing

We will now explore what happens when the sampling frequency, f_s, does not meet the Nyquist criterion. In the **Wavegen** in WaveForms, enter the following configuration:

- **Type**: Sine
- **Frequency**: 15 kHz
- **Amplitude**: 1 V
- **Offset**: 0 V
- **Symmetry**: 50%
- **Phase**: 0°

Run the Wavegen. Open the `CyDAQ Software` and send the following configuration:

- **Sampling Rate**: 20 000 Hz
- **Input**: Analog In
- **Filter**: All Pass

Capture about one second of data using the **Start Sampling** button in the `CyDAQ Software`. Save the data as `dat_15k_at_20k.mat` in your project directory.

Create a new script in your project directory called `test_15k_at_20k.m` and paste the contents of `test_1000_at_2100.m` into it. Modify the script so that the contents of the data signal, `x`, and time vector, `t` are loaded from `dat_15k_at_20k.mat`. Run the script and include both the time domain and frequency spectrum plots produced in your lab report. Describe the apparent frequency spectrum of the sampled signal. How might this change if the input signal was a 18 kHz sinusoid rather than a 15 kHz sinusoid?

11.4 Report Checklist

Work with lab partner(s) to create a lab report for this lab exercise. Be sure to include the names of each person in your group on the report. This report should be a typed document that outlines the basic procedure of the lab exercise and should include any results you were asked to obtain. Your report should follow a *cohesive narrative*—it should always be apparent *why* you have presented the results you have. Do not simply insert results without (1) motivating the problem and (2) providing an analysis of the data. Ensure that your report is grammatically correct with complete sentences. This does not mean your report must be wordy with filler—in fact, avoid filler and state your ideas concisely.

Be sure the following are included in your report:

- Section 11.3: Include both the time domain and frequency spectrum plots produced by `test_5_1k_at_20k.m`.
- Section 11.3: Include both the time domain and frequency spectrum plots produced by `test_50_1k_at_20k.m` in your report. Compare the frequency spectrums produced by the `test_5_1k_at_20k.m` and `test_50_1k_at_20k.m` scripts. Discuss the effect(s) on the frequency spectrum of increasing the number of periods from $N = 5$ to $N = 50$.
- Section 11.3: Include both the time domain and frequency spectrum plots produced by `test_50_8k_at_20k.m` in your report. Compare the frequency spectrums produced by the `test_50_1k_at_20k.m` and `test_50_8k_at_20k.m` scripts. Discuss the effect(s) on the frequency spectrum of increasing the frequency of the sine wave from 1 kHz to 8 kHz.
- Section 11.3.1: Discuss why ideal low-pass filters are difficult to realize and why frequency content close to the edge of the ideal rect function might be more prone to distortion if imperfect realizations of a low-pass filter are utilized.
- Section 11.3.1: Include both the time domain and frequency spectrum plots produced by `test_1000_at_2100.m`. Discuss whether the appearance of the time-domain signal comports with the content of the frequency spectrum.
- Section 11.3.2: Include both the time domain and frequency spectrum plots produced by `upsample_1000_at_2100.m`. Discuss whether the plots of the upscaled reconstruction

comport with the plots produced by `test_1000_at_2100.m`.

- Section 11.3.3: Include both the time domain and frequency spectrum plots produced by `test_15k_at_20k.m` in your lab report. Describe the apparent frequency spectrum of the sampled signal. Discuss how this might change if the input signal was a 18 kHz sinusoid rather than a 15 kHz sinusoid.

11.5 Acknowledgments

This lab exercise is based on an exercise drafted by **Isaac Rex**.

11.A Plot Spectrum MATLAB Code

```
function [X,f] = plot_spectrum(x, Fs)
% PLOT_SPECTRUM Plots the mangitude spectrum of a signal.
%   [X, f] = PLOT_SPECTRUM(x, Fs) plots the magnitude spectrum of a signal, x,
%   sampled at the sampling frequency, Fs.
%
% Outputs
%       X: DFT of signal, x
%       f: Frequency vector of the plot from -Fs/2 to Fs/2 (Hertz)

% Written By: Isaac Rex, 2020
% Updated By: Aaron Fonseca, 2024


    % Compute the DFT
    N = length(x);
    X = fft(x);           % Get the DFT from k = 0:N-1
    X_shifted = fftshift(X); % Shift the DFT so k = -N/2:(N/2-1)
    X_mag = abs(X_shifted); % Get the magnitude of X

    % Get the indices and frequencies
    kk = ceil(-N/2):ceil(N/2-1); % Indices of the DFT. ceil() fixes the
```

```
22                                 % indices when N is odd
23      f = kk*Fs/N; % Convert to frequencies between -Fs/2 and Fs/2
24
25      % Plot
26      clf;
27      stem(f, X_mag, 'LineWidth', 1.5, 'MarkerFaceColor', 'none');
28      xlabel('Frequency (Hz)', 'FontSize', 12);
29      axis([-Fs/2, Fs/2, floor(min(X_mag)), ceil(max(X_mag))])
30      ylabel('Magnitude Of DFT', 'FontSize', 12);
31      title_str = sprintf(['Mag Spectrum ' ...
32          'sampled at %.1f kHz'], Fs/1e3);
33      title(title_str, 'FontSize', 16)
34      grid on;
35  end
```

Demodulation with a Software Defined Radio

An Exploration of Communication Systems

Overview

In this lab exercise, students will use software-defined radios (SDRs) to view, demodulate, and listen to AM and FM signals. Iowa State University's *Electronics and Technology Group* (ETG) will provide students with RTL-SDR peripherals for each lab station. Students will interact with the RTL-SDR peripheral using two separate software components: the `SDRSharp` software (freeware from AIRSPY) and a collection of MATLAB scripts (provided in the *supplementary materials* for this exercise or Appendices 12.A–12.E). Iowa State Amateur Radio Club members will be present during the lab session to transmit a custom AM message at 150 MHz for students to decode.

Learning Objectives

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

1. Use the `SDRSharp` software to view the spectra of FM radio signals.
2. Demodulate and listen to FM and AM stations using `SDRSharp`.
3. Demodulate and listen to AM signals in MATLAB.

Materials

(1×) RTL-SDR Receiver

12.1 Introduction

Software-defined radios (SDRs) are customizable radio systems that implement software-configurable signal processing operations. SDRs can be configured to receive or transmit various modulated message signals using a software peripheral. These highly configurable systems can be easily

reprogrammed to implement unrelated modulation schemes, including AM, FM, and more exotic methodologies. SDRs typically contain a local oscillator whose frequency is tuned to a frequency of interest. Since radio frequencies operate well beyond the sampling rates necessary for representing the demodulated signal, SDRs typically implement a down-conversion process that represents data using a more manageable sampling rate. In addition to demodulating AM and FM radio signals, SDRs can be used for all kinds of additional spectrum operations, including amateur radio,[1] obtaining live airplane position information,[2] receiving satellite imagery from NASA satellites,[3] and implementing a police scanner.[4]

A wide variety of SDRs can be purchased on the open market. The SDR used in this lab is the RTL-SDR,[5] an affordable option popular with hobbyists and supported by a large open-source community. While the RTL-SDR can only receive signals, other SDR models can act as both transmitters and receivers.

12.2 The SdrSharp Software

The software used to configure the RTL-SDR peripheral is `SdrSharp`. This software is freely available for download from the AIRSPY website;[6] however, the software has already been installed at each lab station by the *Electronics and Technology Group* at Iowa State University. Navigate to `SDRSharp.exe` and launch the program.

12.2.1 Using the Software

The Main Window

Once the software is loaded, it should open into the *Main Window*, shown in Fig. 12.1. Refer back to this figure as you complete the following steps.

1. First, confirm that the proper *Source Device* is selected. The proper device should be `RTL-SDR USB`. If not set properly, select `RTL-SDR USB` from the *Source Device* drop-down menu.
2. Now, press the *Start/Stop Radio* toggle to *start* the SDR device to begin outputting noise. You should hear radio static from your PC Speaker (unless you are close to a defined radio

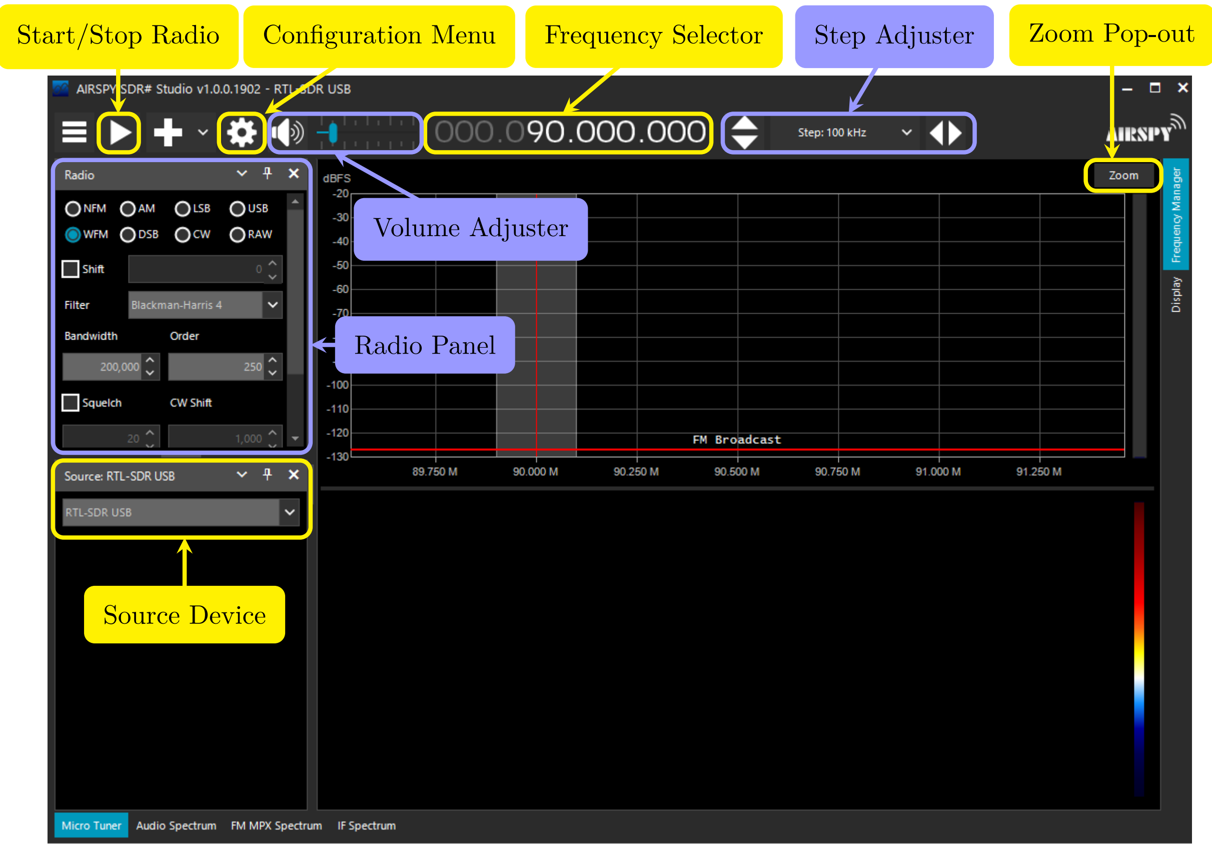


Figure 12.1: The `SdrSharp` Main Window with some of the less prominent features highlighted.

station). The *volume* of the radio output is controlled via the *Volume Adjuster*. If you cannot hear any sound output whatsoever, confirm that your Audio Output Device[1] is set to `Speakers`.

3. The current *tuning frequency* the SDR is displayed in the *Frequency Selector*. This *tuning frequency* can be adjusted using either the *Frequency Selector* or the *Step Adjuster* controls.
    (a) The *Frequency Selector* is the easier of the two controls. Mouse over the digit in the *Frequency Selector* you would like to adjust—and a set of opaque up/down arrows will appear atop that digit. You can then click either of these arrows to shift the value of that digit by ± 1.
    (b) The *tuning frequency* can also be adjusted using the *Step Analyzer* control or by clicking regions within the *main spectrum window*. This method is less intuitive,

[1]Click the Speaker Icon in the Windows System Tray (the rightmost section of the taskbar). You should see a system volume slider and a drop-down available audio devices situated just above it.

and you are encouraged to experiment with adjusting the *tuning frequency* using this method at your leisure.

4. The method/type of *demodulation* is controlled within the *Radio Panel*. The different methods of *demodulation* available are given the set of radio buttons at the top of the panel. These buttons are labeled:
   - `NFM` for *Normal FM*,
   - `WFM` for *Wide FM*,
   - `AM` for *AM*,
   - `DSB` for *Double Sideband AM*,
   - `LSB` for *Lower Sideband AM*,
   - `USB` for *Upper Sideband AM*,
   - `CW` for *Continuous Wave*,
   - and `RAW` for *No Demodulation*.
5. The *bandwidth* of the *demodulation window* is controlled within the *Radio Panel* in the *Bandwidth* selector box. This is only adjustable for certain methods of demodulation.

This covers most of the essential controls visible in the *Main Window*, but a few remain tucked away in the *Zoom Pop-out* panel.

The Zoom Pop-out Panel

This panel is revealed by clicking the *Zoom Pop-out* toggle in Fig. 12.1. The resulting pop-out panel is depicted in Fig. 12.2. Note that the control target associated with each slider within the pop-out panel is given by the label **above** that particular slider. For instance, the uppermost slider controls the *Zoom* control target.

1. The *Zoom* slider (the uppermost slider) is the most important control in this pop-out panel. It controls the *range of frequencies captured* by the SDR; this is essentially the x-axis of the *main spectrum window*. Mess around with the slider and see what happens.
2. The remaining controls: *Contrast*, *Range*, and *Offset*—affect the *secondary spectrogram window*. Mess around with these and see what happens. These controls are not relevant to this lab exercise.

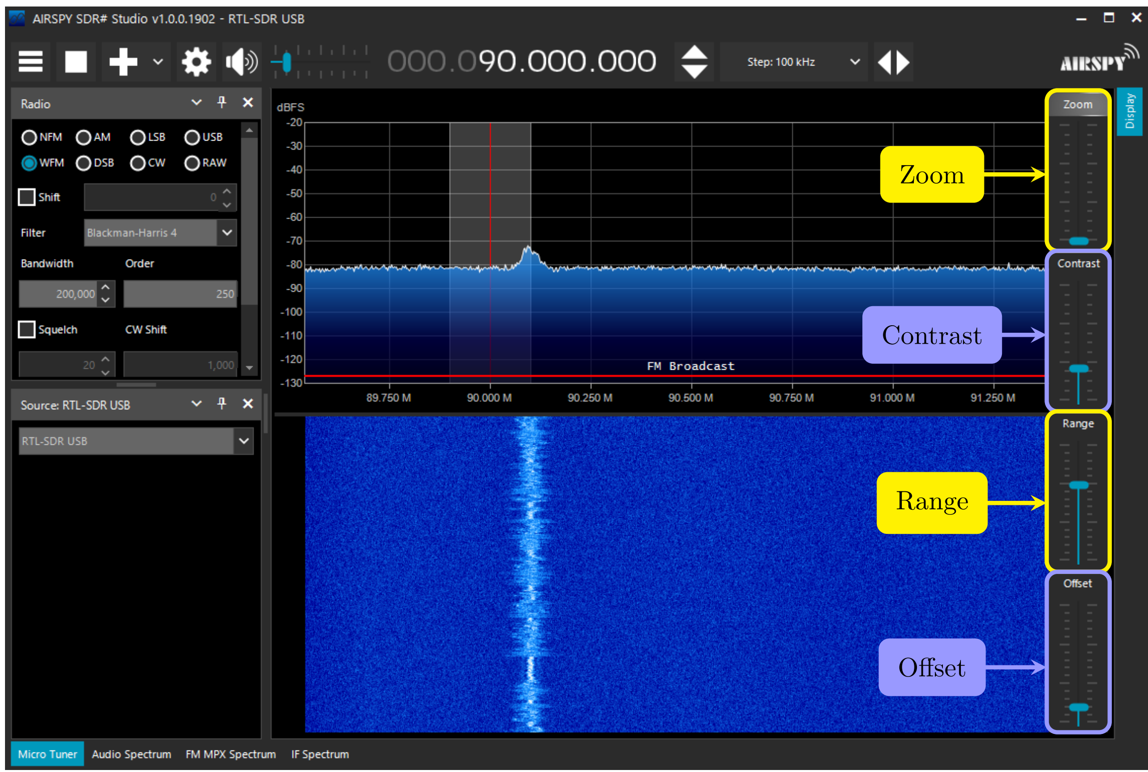


Figure 12.2: The controls within the *Zoom Pop-out* panel; counter-intuitively, each control label is placed *above* its associated slider—this diagram offers more explicit labeling.

There remains an additional yet **critical** control hidden within the *Configuration Window*.

The Configuration Window

This window is revealed by clicking the *Configuration Menu* button (the gear icon depicted in Fig. 12.1). This opens the *Configuration Window*, depicted in Fig. 12.3.

1. First, ensure that *Offset Tuning*, *RTL AGC* and *Turner AGC* are each unchecked.
2. The *gain control* (or RF Gain) is a **critical** control which adjusts the *input gain* to the demodulator. Try adjusting this slider and see what happens. When you've changed demodulation methods, you will *almost certainly* need to adjust this *gain*.

This covers all the windows and panels within the software relevant to this lab exercise.

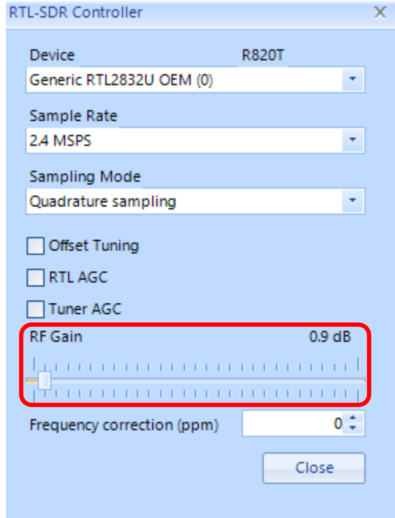


Figure 12.3: The *Configuration Window*; the *gain control* (RF Gain) is highlighted in red.

12.3 Demodulating with SdrSharp

12.3.1 FM Demodulation

This portion of the exercise will focus on viewing the spectra of FM signals and using `SdrSharp` to demodulate and listen to FM stations. First, select the *Wide FM* (e.i., the `WFM`) demodulation method. Now scan the FM broadcasting Band (88–108 MHz) for FM radio stations.

1. Find at least three FM stations and write down their center frequencies. See if you can find ISU's student-run radio station, KURE, or the Iowa Public Radio station, WOI. Feel free to google a list of local FM radio stations. Fig. 12.4 gives an example of one such FM station.
2. Describe the shape of the spectrum of the FM signals you found. Does it bend towards the center frequency like a triangle? Or does it form more of a lump? You will probably need to increase the *gain control* (RF Gain) in the *Configuration Window* to see more than one FM signal; however, increasing the *gain* will also increase the noise introduced.

You might notice that some FM stations have spectra that look different from the others. These stations will have two "boxes" to the left and right of the center frequency. These stations broadcast an "HD Radio" multicast, allowing additional digital stations to be broadcast between traditional analog FM stations. Fig. 12.5 gives an example of an HD station. The HD station(s)

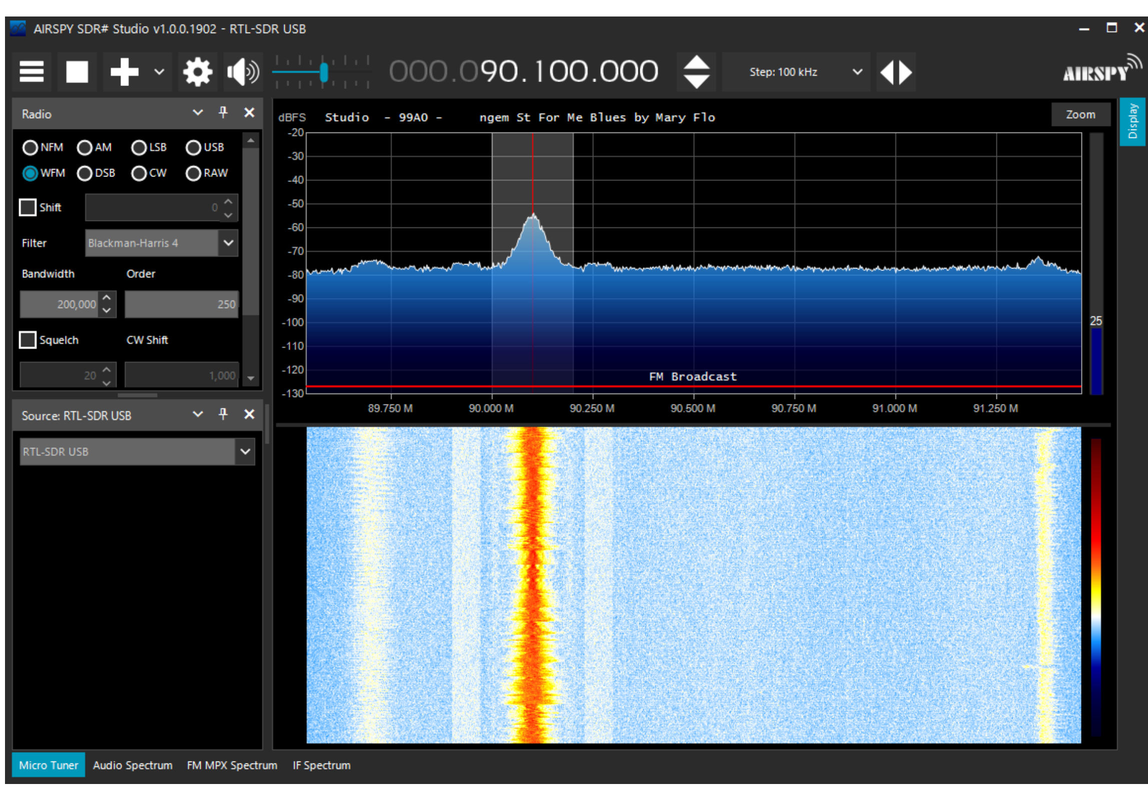


Figure 12.4: Demodulating an FM station in `SdrSharp`.

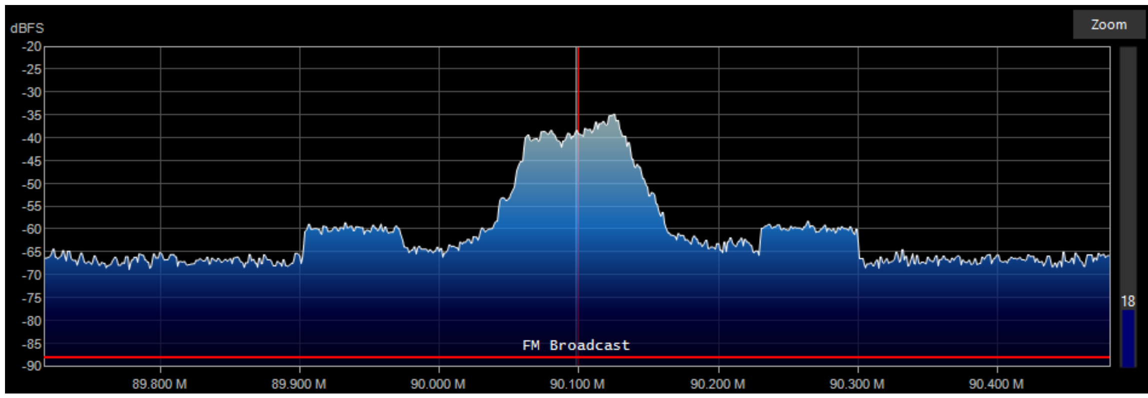


Figure 12.5: The Spectrum of an “HD Radio” station. Note the two box-like protrusions in the spectrum.

receivable in Ames, Iowa might not look nearly as clear as the one depicted in Fig. 12.5. Locate at least one station that has an HD multicast. In your lab report, record the frequency for this station. These can be hard to spot, so make your best guess.

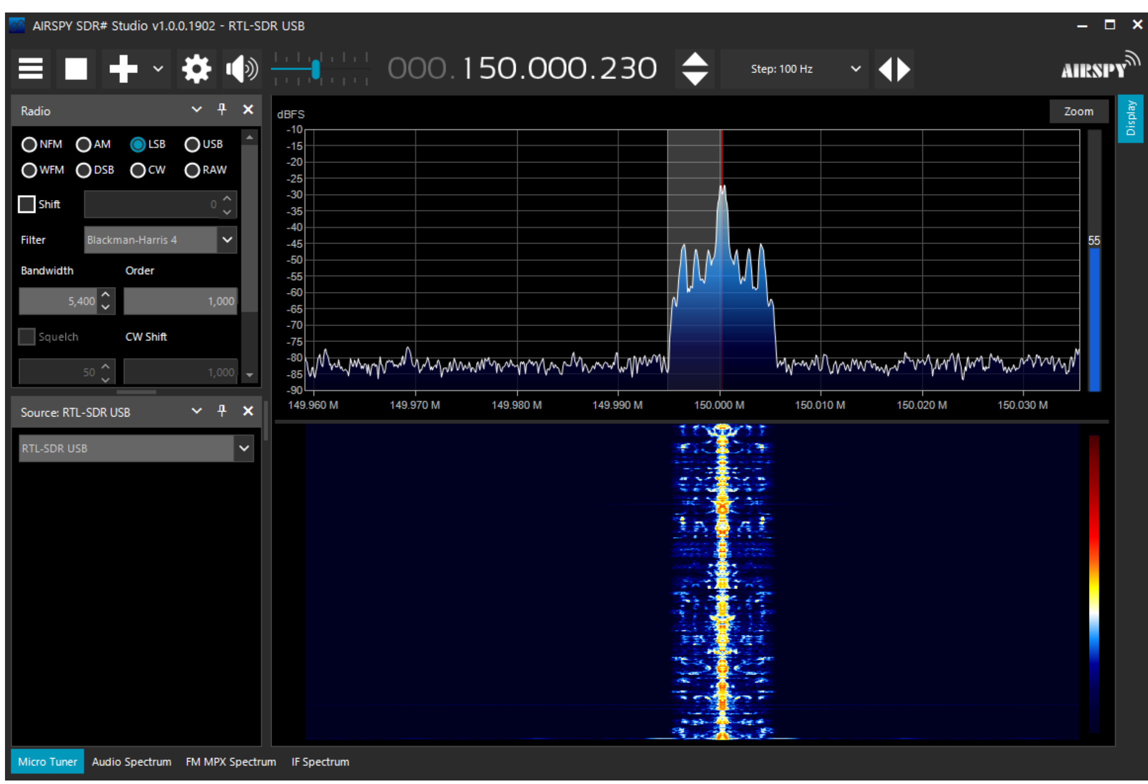


Figure 12.6: Demodulating the in-lab AM station in `SdrSharp`.

12.3.2 AM Demodulation

AM radio is broadcast in the 540–1,700 kHz band, which is too low in frequency for the RTL-SDR to pick up. Fortunately, Iowa State Univerity Amateur Radio Club members will be present in lab to transmit an in-house AM signal in the FM band 150 MHz. Use *Single Side Band AM Demodulation* to demodulate the signal:

1. First, select `LSB` (Lower Sideband AM) demodulation within the *Radio Panel* of the *Main Window*.
2. Next, set the *Bandwidth* (also located within the *Radio Panel*) to approximately 5, 400 Hz.
3. Finally, hone in on the 150 MHz broadcast frequency. You will need to align the red (or right-hand) side of the *demodulation window* to the center point of the AM station spectra.

Fig. 12.6 gives an example of the correct setup for the in-lab AM broadcast. Describe what you hear on the in-house AM station using SDRsharp.

12.4 AM Demodulation with MATLAB

MATLAB's Communications Toolbox provides an interface to the SDR, allowing us to collect samples of the RF signal directly into MATLAB. We will now demodulate the in-house AM transmission using *two* different demodulation strategies, *envelope detection* and *coherent detection*, which are implemented as MATLAB scripts. These scripts can be found in the *supplementary materials* for this exercise or Appendices 12.A–12.E. Place each script within a single folder and open the standalone `amSample_coherent.m` and `amSample_envelope.m` scripts in MATLAB. The remaining `coherent_detect.m`, `envelope_detect.m` and `sdr_read.m` scripts serve as dependencies. Ensure your MATLAB *working directory* is appropriately set to the folder containing each of the scripts. Be sure to close `SdrSharp` before running the MATLAB scripts.

12.4.1 Demodulating using Envelope Detection

We intend to use the scripts to demodulate our signal of interest which lies at about 150 MHz. Within the standalone MATLAB scripts, the F_{center} parameter specifies the tuning frequency while the F_{offset} parameter specifies to degree to which the obtained spectrum is shifted to accommodate a reasonable data rate. Notice that sampling this signal would require a sampling rate of at least 300 *million* samples per second according to the Nyquist-Shannon sampling theorem—eight seconds of audio would require over two billion data samples! To make this more manageable, both scripts down-convert (or frequency shift) the signal retrieved from the SDR by F_{offset} before bringing samples into MATLAB.

1. Open `amSample_envelope.m` in MATLAB.
2. Set F_{offset} (named `F_offset` in the script) to a value between 20 and 100 kHz.
3. Run the script. It takes about 8 seconds to run, so be patient.
4. Once it's complete, it will play the demodulated signal (though it won't necessarily sound the greatest) and produce a plot of the spectrum with respect to time (i.e., a spectrogram).
5. Use this spectrogram to locate the center frequency of the AM signal. Remember that it will be offset by $F_{\text{center}} - F_{\text{offset}}$ from the original carrier frequency. Do this for *two* different F_{offset} values and capture the resulting plots. Note the center frequencies in both cases.

12.4.2 Demodulating using Coherent Detection

We can also use coherent detection to recover the message signal. In this case, we multiply the AM signal, $x(t) = A(1 + k_a m(t)) \cos(\omega t + \theta)$, by a *local oscillator* (LO), $z(t) = \cos(\omega t - \phi)$. Generally, the AM signal's phase angle, θ, is unknown at the receiver since the transmitter and receiver use different oscillators, and the distance between them is unknown. Thus, the received signal is:

$$\begin{aligned} y(t) &= x(t)z(t) \\ &= A(1 + k_a m(t)) \cos(\omega t + \theta) \cos(\omega t + \phi) \\ &= \frac{A}{2}(1 + k_a m(t)) \cos(\theta - \phi) + \frac{A}{2}(1 + k_a m(t)) \cos(2\omega t + \theta + \phi) \end{aligned}$$

The second term, a cosine at twice the carrier frequency, is removed by a lowpass filter, leaving:

$$y_{\text{filt}}(t) = \frac{A}{2}(1 + k_a m(t)) \cos(\theta - \phi)$$

Notice that if $\phi \approx \theta$, $\cos(\theta - \phi) \approx 1$. If θ and ϕ are close to 90° apart, $\cos(\theta - \phi) \approx 0$, and the message will be lost.

1. Open `amSample_coherent.m` in MATLAB.
2. Set LO phase, ϕ, (named `phase_offset` in the script) to a value between 0 and 180 degrees.
3. Run the script. It takes about 8 seconds to run, so be patient.
4. Once it's complete, it will play the demodulated signal (though it won't necessarily sound the greatest).
5. Describe the sound you hear. Do this for *two* different ϕ values.

12.5 Report Checklist

Work with lab partner(s) to create a lab report for this lab exercise. Be sure to include the names of each person in your group on the report. This report should be a typed document that outlines the basic procedure of the lab exercise and should include any results you were asked to

obtain. Your report should follow a *cohesive narrative*—it should always be apparent *why* you have presented the results you have. Do not simply insert results without (1) motivating the problem and (2) providing an analysis of the data. Ensure that your report is grammatically correct with complete sentences. This does not mean your report must be wordy with filler—in fact, avoid filler and state your ideas concisely.

Be sure the following are included in your report:

- Section 12.3.1: Write down three FM stations.
- Section 12.3.1: Describe the shape of FM radio spectra.
- Section 12.3.1: Write down one FM station that uses HD Radio.
- Section 12.3.2: Describe what you hear on the in-house AM station using SDRsharp.
- Section 12.4.1: Describe what you hear using envelope detection. Provide copies of the spectrograms from envelope detection. Run the code for two different F_{offset} values and note the change on the spectrogram.
- Section 12.4.2: Describe what you hear using coherent detection. Run the code for two different phase values and comment on any changes in the signal quality.

12.6 Acknowledgements

This lab exercise is based on an exercise drafted by **Connor Ryan** in co-operation with **Andrew Bolstad**. Hardware support and troubleshooting were provided by **Mathew Post** in co-operation with the *Electronics and Technology Group*.

12.A AM Coherent Demodulation Standalone Script

This standalone script requires the `coherent_detect.m` function (see Appendix 12.C) to run.

Listing 12.1: `amSample_coherent.m`

```
clearvars; close all; clc;
% Coherent Demodulation Standalone Script
% This script acquires the data from the SDR and demodulates it using
% coherent demodulation.
```

```
5
6  % Written By: Andrew Bolstad and Connor Ryan, 2022
7  % Updated By: Aaron Fonseca, 2024
8
9
10 F_center = 150e6; % AM carrier frequency, currently at 150 MHz
11 F_offset = 100e3; % User-configurable parameter from 20 to 100 kSpS
12
13 Fs_sdr = 250e3;                   % Sampling rate of the SDR receiver
14 downsample_ratio = 5;
15 Fs_audio = Fs_sdr / downsample_ratio; % Sampling rate of the demodulated audio
16
17 % Try different phase_offset values to see if you can improve the signal output
18 phase_offset = 0;
19
20 % Read 8 seconds of data from the SDR receiver
21 % (down-converts by F_center - F_offset)
22 sdr_data = sdr_read(8, Fs_sdr, F_center, F_offset);
23
24 % Display spectrum of received signal
25 figure(1);
26 spectrogram(sdr_data, [], [], [], Fs_sdr);
27 title('Spectrum of shifted signal');
28
29 % Demodulate
30 demod_data = coherent_detect(sdr_data, F_offset, Fs_sdr, phase_offset);
31 % Downsampling to a frequency the audio card can handle
32 audio_data = downsample(demod_data, downsample_ratio);
33
34 % Display spectrum of audio signal
35 figure(2);
36 spectrogram(audio_data, [], [], [], Fs_audio, 'centered');
37 title('Spectrum of audio signal');
38
39 % Play audio signal
40 soundsc(audio_data, Fs_audio);
```

12.B AM Envelope Demodulation Standalone Script

This standalone script requires the `envelope_detect.m` function (see Appendix 12.D) to run.

Listing 12.2: `amSample_envelope.m`

```
clearvars; close all; clc;
% Envelope Demodulation Standalone Script
% This script acquires the data from the SDR and demodulates it using
% envelope demodulation.

% Written By: Andrew Bolstad and Connor Ryan, 2022
% Updated By: Aaron Fonseca, 2024

F_center = 150e6; % AM carrier frequency, currently at 150 MHz
F_offset = 70e3; % User-configurable parameter from 20 to 100 kSpS

Fs_sdr = 250e3;                  % Sampling rate of the SDR receiver
downsample_ratio = 5;
Fs_audio = Fs_sdr / downsample_ratio; % Sampling rate of the demodulated audio

% Read 8 seconds of data from the SDR receiver
% (down-converts by F_center - F_offset)
sdr_data = sdr_read(8, Fs_sdr, F_center, F_offset);

% Display spectrum of received signal
figure(1);
spectrogram(sdr_data, [], [], [], Fs_sdr);
title('Spectrum of shifted signal');

% Demodulate
demod_data = envelope_detect(sdr_data, F_offset, Fs_sdr);
% Downsampling to a frequency the audio card can handle
audio_data = downsample(demod_data, downsample_ratio);
```

```
29
30 % Display spectrum of audio signal
31 figure(2);
32 spectrogram(audio_data, [], [], [], Fs_audio, 'centered');
33 title('Spectrum of audio signal');
34
35 % Play audio signal
36 soundsc(audio_data, Fs_audio);
```

12.C AM Coherent Detection Function

This function requires the `sdr_read.m` function (see Appendix 12.E) to run.

Listing 12.3: `coherent_detect.m`

```
1  function audio_data = coherent_detect(rf_data, F_carrier, Fs, phase_offset)
2  % COHERENT_DETECT demodulates AM signals using the coherent detection method
3  % audio_data = COHERENT_DETECT(rf_data, F_carrier, Fs, phase_offset)
4  %
5  % Inputs
6  %   rf_data: a vector containing the am modulated signal.
7  %   F_carrier: a scalar specifying the carrier frequency.
8  %   Fs: a scalar specifying the sampling rate of rf_data.
9  %   phase_offset: a scalar specifying the phase offset of the carrier.
10
11 % Written By: Andrew Bolstad and Connor Ryan, 2022
12 % Updated By: Aaron Fonseca, 2024
13
14
15     if (nargin < 4)
16         phase_offset = 0;
17     end
18
19     t = (0:length(rf_data)-1) / Fs;
20     carrier_data = cos(2*pi*F_carrier*t - phase_offset);
21     message_data = rf_data .* carrier_data.';
```

```
22
23      [B, A] = butter(6, F_carrier * 2/Fs);
24      audio_data = filtfilt(B, A, message_data) * 2;
25
26      % might not be necessary
27      audio_data = audio_data - mean(audio_data);
28  end
```

12.D AM Envelope Detection Function

This function requires the `sdr_read.m` function (see Appendix 12.E) to run.

Listing 12.4: `envelope_detect.m`

```
1   function audio_data = envelope_detect(rf_data, F_carrier, Fs)
2   % ENVELOPE_DETECT demodulates AM signals using the envelope detection method
3   % audio_data = ENVELOPE_DETECT(rf_data, F_carrier, Fs)
4   %
5   % Inputs
6   %   rf_data: a vector containing the am modulated signal.
7   %   F_carrier: a scalar specifying the carrier frequency.
8   %   Fs: a scalar specifying the sampling rate of rf_data.
9
10  % Written By: Andrew Bolstad and Connor Ryan, 2022
11  % Updated By: Aaron Fonseca, 2024
12
13
14      t = (0:length(rf_data)-1) / Fs;
15      rect_signal = rf_data;
16      rect_signal(rect_signal<0) = 0;
17
18      % apply Butterworth filter
19      [B, A] = butter(6, F_carrier * 2/Fs);
20      audio_data = filtfilt(B, A, rect_signal) * 2;
21  end
```

12.E SDR Read Function

Listing 12.5: sdr_read.m

```
function data = sdr_read(Tdur, Fs, F_center, F_offset)
% SDR_READ reads data from the SDR peripheral
%   data = SDR_READ(Tdur, Fs, F_center, F_offset) captures Tdur seconds of data
%   from the SDR peripheral at a sampling rate of Fs with a center frequency of
%   F_center and an offset frequency of F_offset.
%
% Inputs
%   Tdur: a scalar containing the duration (in seconds) of data to capture.
%   Fs: the sampling rate of the captured data.
%   F_center: the center frequency to tune the receiver.
%   F_offset: the offset frequency to configure on the receiver.

% Written By: Andrew Bolstad and Connor Ryan, 2022
% Updated By: Aaron Fonseca, 2024

    rx_frame_size = 2048;        % Size of one frame of rx samples
    F_shift = F_center - F_offset; % Frequency shift (Hz)
    T0 = rx_frame_size / Fs;     % Time length of one sample frame
    num_rx_frames = round(Tdur / T0); % Number of frames received
    rx_gain = 7.7;

    % Initialize the SDR object
    rx_sdr = comm.SDRRTLReceiver('0', ...
                            'CenterFrequency', F_shift, ...
                            'SampleRate', Fs, ...
                            'SamplesPerFrame', rx_frame_size, ...
                            'EnableTunerAGC', false, ...
                            'TunerGain', rx_gain, ...
                            'OutputDataType', 'double');
    rx_data_full = [];
    % Stream data from the device
```

```
32     for p=1:num_rx_frames
33         rx_data = rx_sdr();
34         rx_data_full = cat(1, rx_data_full, rx_data);
35     end
36     data = real(rx_data_full);
37 
38     % Release the SDR object when finished
39     release(rx_sdr)
40 end
```

12.R References

[1] TheSmokinApe. "HF HAM radio with RTL-SDR made easy!" Youtube. (Aug. 2022), [Online]. Available: http://www.youtube.com/watch?v=vCNOjDeOTj4.

[2] "RTL-SDR tutorial: Cheap ADS-B aircraft RADAR," RTL-SDR.COM. (2013), [Online]. Available: http://www.rtl-sdr.com/adsb-aircraft-radar-with-rtl-sdr/.

[3] "SatDump version 1.1.0 released," RTL-SDR.COM. (2013), [Online]. Available: http://www.rtl-sdr.com/satdump-version-1-1-0-released-feature-overview/.

[4] Nate the Robot. "How to set up SDRTrunk part 1 (the basics)," Youtube. (Nov. 2020), [Online]. Available: http://www.youtube.com/watch?v=jy_INdGBTM0.

[5] "RTL-SDR (RTL2832U) and software defined radio news and projects," RTL-SDR.COM. (2023), [Online]. Available: http://www.rtl-sdr.com/.

[6] "SDR software download," AIRSPY. (2024), [Online]. Available: https://airspy.com/download/.