University of Michigan
EECS 206 Laboratory Manual

Mark A. Bartsch David L. Neuhoff Gregory H. Wakefield

First Edition
August 2002

Preface

This laboratory manual was written during the first three semesters that EECS 206 was taught at the University of Michigan. It represents an effort to provide hands-on experience with signals and systems engineering and concepts by working with the MATLAB mathematics environment. The specific goals are:

- To reinforce the learning of the material presented in the lecture.
- To acquaint students with a number of problems/tasks addressed by signals and systems engineering, and with some of the approaches to these problems.
- To involve students in the design, implementation and testing of systems that address some signals and systems engineering tasks.
- To familiarize students with the use of MATLAB as a primary prototyping tool for signals and systems engineering.

While not a "programming" class, it is important that students be able to do things for themselves, such as implement a solution to some basic signals and systems task. In signals and systems engineering, this often involves programming in a language like MATLAB. However, for these laboratories we have attempted to limit the amount of "programming for the sake of programming," which is better obtained in a true programming course. What remains should allow students to gain facility with MATLAB without requiring advanced programming skills.

The lab assignments presume that students have had some significant programming experience, e.g. a first course at the freshman level, and some experience with MATLAB, e.g. two to three weeks of coverage in a first programming course. This prerequisite notwithstanding, the lab manual begins with a tutorial, which serves to review MATLAB and to emphasize the constructs needed in these assignments. It has been found that students with significant programming experience but no prior MATLAB can also succeed in these laboratory assignments, provided they make the extra effort to focus strongly on MATLAB during the first couple of weeks of the course. That is, MATLAB is readily learnable by people familiar with another programming language.

The laboratory assignments are intended to be mostly self-contained. To this end, each contains a substantial amount of background material. This material highlights important theoretical concepts, introductions to specific signals and systems problems/tasks, and the specific approaches to the solution of problems to be examined in the assignments. In some cases, the material in the background section for each lab is meant as a reference rather than as strictly necessary to the completion of the laboratory assignment. In other cases, the background material describes an approach that you will use in the laboratory assignment.

Each lab assignment also contains a MATLAB section introducing commands or techniques that will be important in this assignment, a demo section listing the demonstrations that will take place

in the lab session, and an assignment section listing exactly what must be done. Note that the bullets indicate items to be included in your lab report.

It is highly recommended that you read through each laboratory before arriving at the laboratory session in which you will begin the lab. This will not only give you a better foundation to understand the material in the laboratories, but it will allow you to complete the laboratory more quickly once you have begun working on it.

Commensurate with the first listed goal, all of the laboratories are meant to reinforce key concepts of the course. However, the presentation will often be somewhat different from that of the lecture or textbook. For instance, we develop *convolution* and *filtering* by connecting it to the operation of *correlation*, which we present in the Lab 2. We also use the idea of correlation to motivate the key concepts of *spectrum* and the *Fourier series*. In other cases, we use the visualization capabilities of MATLAB to help develop an intuitive sense of how systems "work." For instance, Lab 9 uses a GUI to graphically show the effects of poles and zeros on the frequency response of a filter.

While the laboratories reinforce material from the lectures and textbook, commensurate with the second and third goals listed earlier, they also go beyond them in numerous places. For instance, the ideas of *detection* and *classification* form a common theme throughout the laboratories. These ideas are not commonly introduced at the undergraduate level, but they form an important component of signals and systems engineering. As another example, Lab 5 develops a transform based image encoder, similar to JPEG. We also focus on the two-dimensional signals (images) in Labs 5 and 6, rather than solely concentrating on one-dimensional signals.

To a great extent, the amount you will get out of these laboratories is dependent upon the amount you put into them. There are a wide variety of topics covered in these labs. We have necessarily not examined them in great depth, but we wish to encourage further thought and exploration into many of them. In many of the labs, you will see items labeled "food for thought." These are exercises that will lead you to examine other aspects of a problem, often in greater depth than in the actual assignment. While these "food for thought" items are in no way required, we strongly recommend that you look at them and discuss them with your lab instructors and peers. Hopefully, you will find many ideas and applications in this course that will interest you and encourage you to explore further.

A note about the "electronic" portion of this laboratory manual. Each laboratory involves the use of MATLAB code, data files, and programs that must be downloaded from the course web page. These programs were developed using MATLAB 6 (Release 12) and a Windows 2000 platform. While most of the code should work on any version of MATLAB, some (most notably the GUI programs) require MATLAB 6 or greater. Additionally, we have provided "compiled" MEX-file versions of many of the programs that you will be writing code to complete. This allows you to check the results provided by your code with "correct" code, and also gives you a way to continue working on the laboratory even if you cannot get the code working. Note, however, that these programs are compiled as Windows .dll files. As such, they will ONLY operate on a Windows-based operating system. In general, we recommend that you use CAEN machines with the latest version of Windows.

Remember that these laboratories are covered by the College of Engineering Honor Code. In particular, it is a violation of the Honor Code to work on these laboratories with others, unless they are members of the lab group to which you are assigned. Further, using, or in any way deriving advantage from, solutions from previous terms is a violation. If you have any questions about how the Honor Code applies to this class, talk to your instructors.

Finally, we would like to acknowledge all of those who helped us during the development of these laboratories. In particular, we would like to thank Professors Stephane Lafortune and Jeffrey Fessler for their input and comments on these laboratories. We would also like to thank the GSIs who

helped us to give the labs a “trial run” during the first three semesters: Norm Adams, Dongsook Kim, Thomas Kragh, Ben Lee, Baptiste Poupard, Charles Hsin, and Fred Zeitz. Finally, we would like to thank the students of EECS 206 during those semesters for their patients and helpful comments during the development and revision of the laboratories.

MAB, DLN, GHW
August 2002

Contents

Preface **iii**

Contents **vi**

List of Figures **xiv**

An Introduction to MATLAB **1**

1 What is MATLAB? . . . 1
- 1.1 MATLAB is a mathematics environment . . . 1
- 1.2 MATLAB is tool for visualizing data . . . 2
- 1.3 MATLAB is a prototyping language . . . 2
- 1.4 MATLAB can do more... . . . 2

2 Demos for the first tutorial lab section . . . 3

3 Using MATLAB: The basics . . . 3
- 3.1 Starting MATLAB . . . 3
- 3.2 How to get help . . . 3
- 3.3 Using MATLAB as a calculator (with variables) . . . 4

4 Vectors, Matrices, and Arrays . . . 5
- 4.1 Constructing arrays . . . 5
- 4.2 Concatenating arrays . . . 5
- 4.3 Transposition and "flipping" arrays . . . 6
- 4.4 Building large arrays . . . 6
- 4.5 The colon operator . . . 6

5 Array Arithmetic . . . 7

6 Indexing . . . 8
- 6.1 Basic indexing . . . 8
- 6.2 Single number indexing . . . 8
- 6.3 Vector indexing . . . 8
- 6.4 Finding the size of an array . . . 9
- 6.5 Vector indexing to modify arrays . . . 9
- 6.6 Conditional statements and the "find" command . . . 9

7 Data Visualization . . . 10
- 7.1 Using "plot" . . . 10
- 7.2 Interpolation; line and point styles . . . 10
- 7.3 Axis labels and titles . . . 10
- 7.4 Commands related to "plot" . . . 11
- 7.5 Plotting with an x-axis . . . 11

7.6 Plotting multiple vectors on the same figure . . . 11
7.7 Legends . . . 12
7.8 Putting several axes on one figure . . . 12
7.9 Two-dimensional arrays . . . 12
8 Programming in MATLAB . . . 13
8.1 Paths and working directories . . . 13
8.2 Types of command files in MATLAB . . . 13
MATLAB scripts . . . 14
MATLAB functions . . . 14
Scripts versus functions . . . 15
8.3 Control Structures . . . 15
Loops . . . 15
Conditional statements . . . 16
8.4 Strings and string output . . . 16
9 Debugging your MATLAB code . . . 17

1 Signals, Signal Statistics, and Signal Detection I **19**
1.1 Introduction . . . 19
1.1.1 "The Questions" . . . 20
1.2 Background . . . 20
1.2.1 Continuous-time and discrete-time signals . . . 20
1.2.2 Describing Signals . . . 21
1.2.3 Signal support and duration . . . 22
1.2.4 Periodicity . . . 23
1.2.5 Signals in MATLAB . . . 23
1.2.6 Signal Statistics . . . 25
1.2.7 Measuring signal distortion and error . . . 27
1.2.8 Signal detection . . . 28
Specifying the detector's operation . . . 29
Detector algorithm . . . 30
1.3 Some MATLAB commands for this lab . . . 31
1.4 Demonstrations in the Lab Session . . . 33
1.5 Laboratory Assignment . . . 33

2 Signal Correlation and Detection II **37**
2.1 Introduction . . . 37
2.1.1 "The Questions" . . . 37
2.2 Background . . . 38
2.2.1 Correlation . . . 38
2.2.2 Running correlation . . . 39
2.2.3 Using correlation for signal detection . . . 40
2.2.4 Using correlation for detection of signals transmitted simultaneously with other signals . . . 41
2.2.5 Noise, detector errors, and setting the threshold . . . 43
2.2.6 An algorithm for running correlation . . . 45
2.3 Some MATLAB commands for this lab . . . 46
2.4 Demonstrations in the Lab Section . . . 47
2.5 Laboratory Assignment . . . 47

3 Sinusoids and Sinusoidal Correlation **51**
3.1 Introduction . . . 51
3.1.1 "The Question" . . . 52
3.2 Background . . . 52
3.2.1 Complex numbers . . . 52
3.2.2 Sinusoidal and complex exponential signals in continuous time . . . 53
3.2.3 Finding the amplitude and phase of a sinusoid with known frequency . . . 53
The Amplitude and Phase Calculator . . . 55
3.2.4 Determining the frequency of a target sinusoid . . . 57
A frequency estimation algorithm . . . 58
Estimating doppler shift . . . 59
3.3 Some MATLAB commands for this lab . . . 59
3.4 Demonstrations in the Lab Section . . . 61
3.5 Laboratory Assignment . . . 61

4 Fourier Series and the DFT **65**
4.1 Introduction . . . 65
4.1.1 "The Questions" . . . 66
4.2 Background . . . 66
4.2.1 Frequency-domain representations . . . 66
4.2.2 Periodic Continuous-Time Signals — The Fourier Series . . . 67
Partial Series . . . 68
T-Second Fourier Series . . . 68
Aperiodic Continuous-Time Signals . . . 69
Properties of the Fourier Coefficients . . . 69
4.2.3 Periodic Discrete-Time Signals — The Discrete Fourier Transform . . . 71
N-point DFT . . . 73
Aperiodic Discrete-Time Signals . . . 74
Approximating Fourier series coefficients with the DFT . . . 74
Properties of the DFT coefficients . . . 74
4.2.4 Separating Signals Based on Differing Harmonic Series . . . 77
4.3 Some MATLAB commands for this lab . . . 78
4.4 Demonstrations in the Lab Section . . . 79
4.5 Laboratory Assignment . . . 80

5 Images, Compression, and Coding **85**
5.1 Introduction . . . 85
5.1.1 "The Question" . . . 86
5.2 Background . . . 86
5.2.1 Images . . . 86
5.2.2 Signal statistics for images . . . 86
5.2.3 Data compression . . . 88
5.2.4 Transformation . . . 89
5.2.5 Quantization . . . 91
5.2.6 Binary coding . . . 91
5.2.7 Performance . . . 92
5.2.8 Designing a transform coder . . . 94
5.3 Some MATLAB commands for this lab . . . 95

5.4 Demonstrations in the Lab Section . . . 98
5.5 Laboratory assignment . . . 98
Postscript: JPEG Compression . . . 101

6 FIR Filtering and Image Processing 103
6.1 Introduction . . . 103
6.1.1 "The Question" . . . 103
6.2 Background . . . 104
6.2.1 Implementing FIR Filters . . . 104
6.2.2 Edge effects and delay . . . 105
6.2.3 Noise and distortion . . . 107
Nonlinear filtering . . . 108
6.2.4 Filtering two-dimensional signals . . . 108
6.2.5 Image processing with FIR filters . . . 109
6.3 Some MATLAB commands for this lab . . . 110
6.4 Demonstrations in the Lab Section . . . 113
6.5 Laboratory Assignment . . . 113

7 Decoding DTMF: Filters in the Frequency Domain 119
7.1 Introduction . . . 119
7.1.1 "The Question" . . . 119
7.2 Background . . . 120
7.2.1 DTMF signals and Touch Tone™ Dialing . . . 120
7.2.2 Decoding DTMF Signals . . . 120
Step 1: Bandpass Filters . . . 121
Step 2: Determining filter output strengths . . . 122
Step 3: "Detect and Decode" . . . 123
7.2.3 Decoder Robustness . . . 124
7.2.4 Sidenote: Searching Parameter Spaces . . . 125
7.3 Some MATLAB commands for this lab . . . 125
7.4 Demonstrations in the Lab Section . . . 127
7.5 Laboratory Assignment . . . 127

8 Classification and Vowel Recognition 133
8.1 Introduction . . . 133
8.1.1 "The Question" . . . 134
8.2 Background . . . 134
8.2.1 An Introduction to Classification . . . 134
8.2.2 A classification example . . . 135
8.2.3 A few more classification examples . . . 138
8.2.4 Formalizing the feature classifier . . . 138
8.2.5 Measuring the performance of a classifier . . . 140
Data usage when designing classifiers . . . 141
8.2.6 Vowel Classification . . . 141
8.3 Some MATLAB commands for this lab . . . 143
8.4 Demonstrations in the Lab Section . . . 145
8.5 Laboratory Assignment . . . 145

9 Filter Design, Modeling, and the z-Plane **149**
9.1 Introduction . . . 149
9.1.1 "The Question" . . . 149
9.2 Background . . . 150
9.2.1 Filters and the z-transform . . . 150
9.2.2 FIR Filters and the z-transform . . . 151
9.2.3 IIR filters and rational system functions . . . 152
Poles and zeros at the origin and at infinity . . . 154
Pole-zero plots . . . 155
9.2.4 Graphical interpretation of the system function . . . 155
9.2.5 Poles and stability . . . 157
9.2.6 Filter design using manual pole-zero placement . . . 157
9.2.7 Design of Standard Filter Types . . . 159
9.2.8 Modeling Vowel Production . . . 159
All-pole analysis and vowel classification . . . 161
9.3 Some MATLAB commands for this lab . . . 162
9.4 Demonstrations in the Lab Section . . . 165
9.5 Laboratory Assignment . . . 166

List of Commonly Used MATLAB Commands **169**

10 Useful MATLAB Facts **170**

List of Figures

1.1 Examples of continuous-time and discrete-time signals 21
1.2 Signal value distribution and a discrete histogram approximation 27
1.3 An "overview" block diagram for a "signal present" detector. 28
1.4 A detailed block diagram for the "signal present" detector. 29

2.1 Examples of positively correlated, uncorrelated, and anticorrelated signals. 38
2.2 Signals in a radar detection system 40
2.3 A generalized block diagram for a correlation-based detection system. 41
2.4 Code signals for a simultaneous communication system 42
2.5 Example of several overlapping communications signals 42
2.6 Example of a MATLAB figure with subplots. 46

3.1 Three-dimensional plot of a complex exponential signal. 54
3.2 System diagram for the "amplitude and phase calculator." 56
3.3 System diagram for the "frequency, amplitude, and phase estimator." 59

4.1 Time-domain and frequency-domain representations 66
4.2 A comparison of the Fourier Series and DFT coefficients 76
4.3 The DFT of a harmonic series 77

5.1 A block diagram of a general data compression system. 85
5.2 A block diagram of a general image encoder/compressor. 88
5.3 A block diagram of a general image decoder/decompressor. 89
5.4 Block diagram of a direct quantizer 92
5.5 Plot of the tradeoff between coding rate and distortion 93
5.6 A block diagram of a transform coder 94

6.1 A graphical illustration of filtering 105
6.2 Input and output of a 5-point moving average filter. 106
6.3 A block diagram of a noise-reduction system 107
6.4 The coefficients of a two-dimensional moving average filter 109
6.5 The coefficients for `g_smooth` filters with varying widths. 112

7.1 A spectrogram of a DTMF signal 121
7.2 A block diagram of the DTMF decoder system 122
7.3 A comparison of half-wave and full-wave rectification 123
7.4 An illustration of the DTMF detector subsystem 124

8.1 Block diagram of a general classifier system. 134
8.2 A histogram for one-feature classification 135
8.3 A case where two-feature classification is necessary 136
8.4 An example where two features are not as clearly separated. 138
8.5 Various classification examples 139
8.6 Block diagram of a distance-based feature classifier 140
8.7 The magnitude spectrum (in decibels) of four vowel signals 142

9.1 A pole-zero plot of an IIR filter. 155
9.2 System function, frequency response, and pole-zero plot of an FIR filter 156
9.3 System function, frequency response, and pole-zero plot of an all-pole filter 156
9.4 System function, frequency response, and pole-zero plot of a general IIR filter . . . 156
9.5 An illustration of the various bands of a lowpass filter. 158
9.6 A diagram showing the larynx and vocal tract. 160
9.7 A block diagram of the source-filter model of speech production. 160
9.8 Plot of the spectral evolution of a vowel signal 161
9.9 The GUI window for *Pole-Zero Place 3-D*. 163

An Introduction to MATLAB

1 What is MATLAB?

The Mathworks, Inc., makers of MATLAB, claims that MATLAB is "the language of technical computing." By and large, they are right. MATLAB is widely used in a great number of scientific fields. For those who work with signals and systems, MATLAB is a de facto standard. Engineers from a wide array of disciplines, in both academia and in industry, use MATLAB on a regular basis. As such, a knowledge of MATLAB will not only be useful for this course, but for future courses and in your career as a whole. One of the main reasons for MATLAB's popularity arises from its wide array of uses. So what is MATLAB?

1.1 MATLAB is a mathematics environment that can easily handle vectors and matrices

MATLAB was originally written to provide an easy-to-use interface to the mathematical subroutines included in LINPACK and EISPACK. These two packages are sets of subroutines written in FORTRAN for a wide variety of linear algebra operations. MATLAB's original focus on linear algebra means that it has very well developed capabilities for handling *vectors* and *matrices*[1]. In fact, MATLAB is short for "*Mat*rix *Lab*oratory." For our purposes, both vectors and matrices are examples of *signals* – a mathematical environment that can easily handle vectors and matrices makes working with signals just as easy.

Let's look at an example to see exactly what this buys us. Suppose that we have two signals, `x` and `y`, each of which is simply an array with 100 elements. How would we add these signals in a language like C++? The easiest way probably involves the following fragment of code:

```
double z[100];
for(int i = 0; i < 100; i++)
{
   z[i] = x[i] + y[i];
}
```

This is a simple enough piece of code, but it is not as clear as it could be. In MATLAB we can simply do the following:

```
z = x + y;
```

[1] Vectors and matrices are simply one- and two-dimensional arrays, respectively

Simply adding two signals (vectors or matrices) with the same size automatically performs an element-by-element sum. Which of these two is easier to understand? Using this MATLAB syntax, we can see immediately what is happening. MATLAB takes care of any necessary looping and variable declarations for us. This is a very common feature in MATLAB; many operations that you would normally need to perform explicitly in another programming language can be performed implicitly in MATLAB.

1.2 MATLAB is tool for visualizing data

You are probably very familiar with how much easier it is to interpret a graph than a table of numbers or a formula. By producing a plot of the relevant data or formula, you can gain a visual sense of what is going on that otherwise might be lacking. This is one of the motivations behind the use of graphing calculators in high school math. Simply put, MATLAB is one of the best tools for visualizing data that is currently available.

You will find that these capabilities very useful in your study of signals and systems. By looking at a signal, you can often gain some insight into how it behaves. The same applies to systems. Certain systems are said to "smooth" signals because of the visual appearance of the resulting signal. In certain cases (like image processing), the visual result of a system is the primary reason for its use.

1.3 MATLAB is a prototyping language

In many respects, MATLAB is like a UNIX shell. It has the same sort of interactive interface for normal usage, but it also has most of the standard programming language constructs like loops and conditional statements. You can put commands into a file and call it as a script. Alternatively, you can write functions with input and output parameters.

The main difference between MATLAB and programming languages like C++ is the ease with which you can implement algorithms (especially mathematical algorithms). This is because MATLAB operates at a higher level than many other programming languages. It is also usually easier to understand MATLAB code than code in other programming languages. The sum-of-vectors example given above is a prime example of this. All of this makes MATLAB a very good *prototyping language*. It is easy to whip up a "proof of concept" program in MATLAB to make sure that your algorithm actually works. Then, you can code a "development" version using a more traditional compiled programming language.

1.4 MATLAB can do more...

One of the key rules of thumb to remember about MATLAB is that it can perform almost any mathematical task you could want. Often, there will be a built-in function to do what you want. If it's not a part of the main MATLAB distribution, it is probably available as part of an add-on called a *toolbox*. Some toolboxes can be purchased from the Mathworks, while others are developed and distributed for free by third party developers.

In this course, we will be focusing on the core MATLAB distribution and the Signal Processing Toolbox. (We will also be doing some image processing, but you will not need the Image Processing Toolbox for this course.) We recommend that you consider purchasing a version of MATLAB and the Signal Processing Toolbox; you find it to be useful throughout your academic career.

2 Demos for the first tutorial lab section

1. Recording, displaying, and manipulating signals in MATLAB
2. Image Compression via JPEG
3. DTMF (Touch-tone) telephone tones

3 Using MATLAB: The basics

3.1 Starting MATLAB

The first step to using MATLAB is to bring up the program on your computer system. This series of laboratories was designed for Windows-based computers, so we recommend using these machines if possible. Starting MATLAB on a Windows machine or a Macintosh usually requires finding the appropriate icon either on the desktop or in the Start menu[2]. At a UNIX system, simply typing "matlab" should be sufficient. Note that you can run MATLAB remotely on UNIX servers through telnet or ssh, but MATLAB versions 6 and higher generally require an X-windows connection to run[3]. When MATLAB is finished loading, you'll see the MATLAB program window, possibly with several subwindows. The most important window is the command window, which contains a command prompt that looks something like this:

```
>>
```

3.2 How to get help

So now what do you do? Well, the first step is to make use of MATLAB's single most useful command:

```
>> help
```

See that list of categories? You can call help on any of these categories to get an organized list of commands with brief discussions. Then, you can call `help` on any of the commands for a complete description of that command. The description also includes a "see also" line near the bottom which suggests other commands that may be related to the one you're looking at. Select a category that looks interesting and call `help` on it. Do the same for whichever command strikes your fancy. For instance:

```
>> help elfun
>> help abs
```

Most often you'll use `help` in this last capacity. Note that `help abs` lists commands related to the absolute value function as well.

Unfortunately the traditional help system isn't so helpful if you don't know the name of the command you're looking for. One way around this is to use the `lookfor` command. For instance, if you know you're looking for a function that deals with time, you can try:

```
>> lookfor time
```

[2] CAEN machines may have multiple versions installed; you should try to locate the most recent version of MATLAB.
[3] Versions of MATLAB prior to 6.x run by default in a terminal window, without an X-windows connection.

This searches the first line of the every help description for the word "time." This can take a while, though (depending upon your system's configuration). You should get into the habit of reading the `help` on every new command that you run across. So call `help` on both `help` and `lookfor`. There's some useful information there.

Another very good source of help is the MATLAB `helpdesk`. It may or may not be available on your system; to find out, simply try:

```
>> helpdesk
```

If it is available, you will see a help window. The MATLAB `helpdesk` contains all of the help pages that you can find using `help` or `lookfor`, along with many other useful documents. The `helpdesk` is also easily searchable (and often much faster than `lookfor`), so you would benefit from becoming familiar with this tool.

3.3 Using MATLAB as a calculator (with variables)

Not surprisingly, you can use MATLAB to do arithmetic. It operates very much like you might expect, employing infix arithmetic like that used on standard calculators. MATLAB can evaluate simple expressions or arbitrarily complicated ones with parentheses used to enforce a particular order of operations.

```
>> 6 * 7
>> (((12 + 5) * 62/22.83) - 5)^2.4
```

(The ^ operator performs exponentiation.) Notice that when you execute these commands, MATLAB indicates that `ans = 7.4977e+003` (or whatever the answer is). This indicates that the result has been stored in a variable called `ans`. We can then refer to this quantity like this:

```
>> 0 * ans
>> ans + 1
```

It is important to note that each of these commands overwrites `ans`. If we want to save an answer, we can simply perform assignment, like this:

```
>> my_variable = 42
```

This is the only declaration of `my_variable` that is needed, and we can use this variable later just as we could with `ans`. Further, `my_variable` will retain its value until we explicitly assign something else to it.

We can also remove variables with the command `clear`. Typing `who` or `whos` will list what variables we have in our workspace.

Using variables, then, is straightforward.

```
>> x = 5.4
>> y = 2
>> z = (my_variable*y)^x
```

Note that sometimes you don't need or want to see what MATLAB returns in response to a particular command. To suppress the output, simply add a semicolon, `;`, after the command. Try any of the above commands with and without the semicolon to see what this does.

We also have access to a wealth of standard mathematical functions. Thus, we can if we want to calculate the sine of the square-root of two and store it in a variable called `var`, we simply type:


```
>> var = sin(sqrt(2))
```

Type `help elfun` to see how to call most of the elementary mathematical functions like these.

There are also a number of constants built into MATLAB that are very useful. The number π is referred to as `pi` (note that MATLAB is case sensitive!). Both `i` and `j` default to $\sqrt{-1}$, but you can still use either (or both) as variable names if you like. You should glance at `help i` so that you can see the various options for building complex numbers. Note that you can overwrite variables like `pi`, `i`, and `j`, but then you will not be able to use their special properties. The special variables (and matrices) built-in to MATLAB are listed under `help elmat`.

4 Vectors, Matrices, and Arrays

So far, we've been using MATLAB to deal with *scalar* numbers. The real power of MATLAB, though, comes from its ability to handle *vectors* and *matrices*. In MATLAB, vectors and matrices are simply one-dimensional and two-dimensional *arrays*, respectively. An array is simply a collection of numbers, each of which is *indexed* by some ordered set of numbers. For instance, a vector is indexed by a single integer, while a matrix is indexed by an ordered pair. The number of indices is equal to the dimension of the array. For instance, consider the following vector and matrix:

$$\mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \qquad M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \tag{1}$$

To access the 3 from vector $\mathbf{v}$, we simply need to know that it is in the third row. (In MATLAB, we use `v(3)` to access this element.) Thus the vector is one-dimensional. To access the 6 in the matrix M, though, we need to know that it is in the second row and the third column. We index the 6 using the pair (2,3), and so matrix is two-dimensional. (In MATLAB, we use `M(2,3)` to access this element.) MATLAB arrays can have any number of dimensions. In practice, though, we will only need vectors and matrices.

4.1 Constructing arrays

There are many different ways to build up and manipulate arrays in MATLAB. For instance, consider (and execute) the following commands:

```
>> a = [1 2 3 4 5 6 7]
>> b = [1, 2, 3, 4, 5, 6, 7]
>> c = [1; 2; 3; 4; 5; 6; 7]
>> d = [1 2 3 4; 5 6 7 8; 9 10 11 12]
```

The first two commands both build up the same vector, a 1×7 *row-vector*[4]. The third command builds up a 7×1 *column-vector* with the same elements. The fourth command builds a 3×4 matrix.

4.2 Concatenating arrays

The comma (or the space) within the square brackets concatenates horizontally and the semicolon concatenates vertically. The elements being concatenated do not need to be scalars, either:

[4] In MATLAB, indices are given as row $\times$ column.


```
>> e = [a b]
>> f = [a; b]
>> g = [c d]
```

Oops! That last command produced an error. When concatenating arrays, the concatenated arrays must have sizes such that the resulting array is rectangular.

4.3 Transposition and "flipping" arrays

The single apostrophe, `'`, is MATLAB's transposition operator. It will turn a row-vector into a column-vector and vice versa. Similarly, it will make an $n \times m$ matrix into an $m \times n$ matrix. To see how this works, type `d'` and look at the results. (Warning: `'` is actually a complex conjugate transpose, so complex numbers will have the sign of their imaginary parts changed. To perform a straight transposition, use the `.'` operator. For real arrays, both operators are identical.) Other useful commands for matrix manipulation include `flipud` and `fliplr`, which mirror matrices top-to-bottom and left-to-right, respectively. Look at `help elmat` for other useful functions.

4.4 Building large arrays

Building small arrays by hand is fine, but it can become very tedious for larger arrays. There are a number of commands to facilitate this. The `ones` and `zeros` commands build matrices that are populated entirely with ones or zeros. The `eye` command builds identity matrices. `repmat` is especially useful for making matrices out of vectors. `diag` builds diagonal matrices from vectors, or returns the diagonal (vector) of a matrix. Check the `help` for all of these commands. For an example, try these:

```
>> ones(5,3)
>> zeros(3,4)
>> zeros(5)
>> eye(4)
```

4.5 The colon operator

The colon operator is one way of creating long vectors that are useful for indexing (see the next section). Execute the following commands:

```
>> 1:7
>> 1:2:13
>> 0.1:0.01:2.4
```

Each of these commands defines a row-vector. With only two arguments, as in the first command, the colon operator produces a row vector starting with the first argument and incrementing by one until the second argument has been reached. The optional middle argument (seen in the second two commands) provides a different increment amount. The colon operator is extremely useful, so it is recommended that you check out `help colon` for more details. Play with some other combinations of parameters to familiarize yourself with the behavior of this operator.

5 Array Arithmetic

MATLAB allows you to perform mixed arithmetic between scalars and arrays as well as two different types of arithmetic on arrays. Mixed scalar/array arithmetic is the most straightforward. Adding, subtracting, multiplying or dividing a scalar from an array is equivalent to performing the operation on every element of the array. For instance,

```
>> [5 10 15 20]/5
```

returns the vector `[1 2 3 4]`.

It is also useful to note that most of the provided mathematical functions (like `sqrt` and `sin`) operate in a similar element-by-element fashion. Thus, the commands

```
>> t = 0:.1:pi;
>> sin(t)
```

return a 32-element vector (the same size as `t`) containing the sine of each element of `t`.

If we have two arrays, addition and subtraction is also straightforward. Provided that the arrays are the same size, adding and subtracting them performs the operation on an element-by-element basis. Thus, the (3,4) element in the output (for instance) is the result of the operation being performed on the (3,4) elements in the input arrays. If the arrays are *not* the same size, MATLAB will generate an error message.

For multiplication, division, exponentiation, and a few other operations, there are two different ways of performing the operation in question. The first involves matrix arithmetic, which you may have studied previously. You may recall that the product of two matrices is only defined if the "inner dimensions" are the same; that is, we can multiply an mxn matrix with an nxp matrix to yield an mxp matrix, but we cannot reverse the order of the matrices. Then, the (p,q) element of the result is equal to the sum of the element-by-element product of the p^{th} row of the first matrix and the q^{th} column of the second. Division and exponentiation are defined with respect to this matrix product. It is not imperative that you recall matrix multiplication here (most likely you will see it in a linear algebra course in the future); however, it *is* important that you note that in MATLAB the standard mathematical operators (`*`, `/`, and `^`) default to these forms of the operations.

A form of multiplication, division, and exponentiation for arrays that is more useful for our purposes is the element-by-element variety. To use this form, we must use the "dot" forms of the respective operators, `.*`, `./`, and `.^`). Once again, the arrays must have the same dimensions or MATLAB will return an error. Thus, the commands

```
>> [1 2 3 4].*[9 8 7 6]
>> [7; 1; 4]./(1:3)'
>> [5 6 7].^[2 3 4]
>> 2.^[1 2 3 4 5 6]
```

perform element-by-element multiplication, division, and two slightly different forms of exponentiation. Note that the `.^` form is necessary even for scalar-to-array exponentiation operations.

The array arithmetic capabilities of MATLAB contribute greatly to its power as a programming language. Using these operators, we can perform mathematical operations on hundreds or thousands of numbers with a single command. This also has the side effect of simplifying MATLAB code, making it shorter and easier to read (usually).

6 Indexing

6.1 Basic indexing

To make arrays truly useful, we need to be able to access the elements in those arrays. First, let's fill a couple of arrays:

```
>> a = 5:5:60
>> d = [9, 8, 7, 6 ; 5, 4, 3, 2]
```

Now, let's access elements in them:

```
>> a(6)
>> a(3) = 12
>> d(2,3)
```

The first command retrieves the sixth element from the vector `a`. The second assigns a number to the third element of the same vector. For the third command, the order of the dimensions is important. **In MATLAB, the first dimension is *always* rows and the second dimension is *always* columns.** Note particularly that this is the opposite of (x, y) indexing. Thus, the third command retrieves the element from row two, column three.

6.2 Single number indexing

We can also index into matrices using single numbers. In this case, the numbers count *down the columns*. This is called "column-major" and is the opposite of array indexing in C or C++. For instance, notice what happens when you use the following commands:

```
>> d(2)
>> d(3)
>> d(7)
```

6.3 Vector indexing

It is not necessary to index arrays only with scalars. One of the most powerful features of MATLAB is the ability to use one array to index into another one. For instance, consider the following commands:

```
>> a([1 4 6])
>> b(3:7)
>> c(2:2:end)
```

These commands return a subset of the appropriate vector, as determined by the indexing vector. For instance, the first command returns the first, fourth, and sixth elements from the vector `a`. Notice the use of the `end` keyword in the third command. In an indexing context, `end` is interpreted as the length of the currently indexed dimension. This is particularly useful because MATLAB will return an error if you try to access the eighth element of a seven-element vector, for instance. In general, indices must be strictly positive integers less than the length of the dimension being indexed. **Thus, unlike C or C++, the indices begin at one rather than at zero.**

Using multiple indices into multi-dimensional arrays is more complicated than doing so with vectors, but in some cases it can be extremely useful. Consider the following commands:


```
>> d([1 3],2)
>> d([2 3],[1 4])
>> d(2,:)
```

The first command, as you might expect, returns the first and third elements of the second column. The second command returns the second and third rows from the first and fourth columns. Note particularly that this command does *not* return the individual elements at (2,1) and (1,4). (To index individual elements in this manner, we need to use single-index method along with the `sub2ind` command). The colon operator in the second command is a shortcut for `1:end`; thus, the third command returns all of the second row.

6.4 Finding the size of an array

Two very useful commands that can be used to facilitate indexing are `size` and `length`. `size` returns a vector containing the length of each dimension of an array. Alternately, `size` can be used to request the length of a single dimension. `length` is primarily useful for vectors when you're not sure about their orientation. `length` returns the length of the longest dimension. Thus, `length(v)` is the same whether `v` is a row-vector or a column-vector, but `size(v,1)` will only properly return the length of a column-vector.

6.5 Vector indexing to modify arrays

It is important to note that all of these indexing techniques are used not only to retrieve many elements from an array but also to set them. When performing array assignment, you must be careful to make sure the array being assigned has the same size as the array to which it is being assigned. For instance, consider the following command:

```
>> d([1 3],[2 4]) = [9 8; 7 6]
```

Note that both of the matrixes on the left and right of the equal sign are 2×2, so the assignment is valid. Look at the results of this command and make sure you understand what it does and why.

6.6 Conditional statements and the "find" command

One last command that is extremely useful in context of indexing in MATLAB is `find`. `find` will return a vector containing the indices of any nonzero elements in an array. Note that `find` uses the single-index indexing scheme that was mentioned earlier. At first glance, this has relatively few uses; however, it is in fact extremely useful because of the behavior of conditional statements in MATLAB (i.e., >, <, and ==). The command `a > 5` will return an array with the same size as `a`, but with each element either 1 or 0 depending on whether or not it is greater than 5. Using `find` on this array will provide the indices of elements greater than 5. One particularly good use of the `find` command is the following contexts. Suppose you wish to set all negative elements in a matrix to zero. You can do this with a single command like so:

```
>> m = [-1 5 10; 3 -8 2; -4 -9 -1];
>> m(find(m < 0)) = 0;
```

Alternately, if you wish to square every element that is greater or equal to 4, you can use the find command twice in a single line, like this:

```
>> m(find(m >= 4)) = m(find(m >= 4)).^2;
```


7 Data Visualization

7.1 Using "plot"

So now we know how to build arbitrarily large arrays, populate them with interesting things, and get individual elements or sections of them. However, pouring over pages and pages of numbers is generally not much fun. Instead, we would really like to be able to visualize our data somehow. Of course, MATLAB provides many options for this. Let's start by building a vector we can use throughout this section, and then looking at it. Execute the following commands:

```
>> x = sin(2*pi*(1:200)/25);
>> plot(x);
>> zoom on;
```

The first command builds up a sine wave, and the second command plots it. A window should have popped up with a sine wave in it. Notice the y-axis extents from -1 to 1 as we would expect. Using this form of `plot`, the x-axis is labeled with the index number; that is, our vector has 200 elements, and so the data is plotted from 1 to 200. The third command turned on MATLAB's zooming capabilities. If you left-click on the figure, it will zoom in; right-clicking[5] will zoom out. You can also left-click and drag to produce a zoom box that lets you control where the figure zooms. Experiment with this zoom tool until you're comfortable with it. Depending on the version of MATLAB that you are using, there may also be an icon of a magnifying glass with a + in it above the figure; clicking this icon will also enable and disable zoom mode.

7.2 Interpolation; line and point styles

If you zoomed in closely enough on the plot, you probably noticed that the signal isn't perfectly smooth. Instead, it is made up of line segments. This is because our vector, `x`, is made up of a finite collection of numbers. MATLAB defaults to *interpolating* between these points on the plot. You can tell MATLAB to show you where the data points are, or to not interpolate, by changing the line and point styles. Try each of these commands and look at the results before executing the next one:

```
>> plot(x,'x-')
>> plot(x,'o')
>> plot(x,'rd:')
```

`help plot` lists the various combinations of characters that you can use to change line styles, point styles, and colors.

7.3 Axis labels and titles

Often, we want to indicate what each axis of a plot represents or add a figure title. The commands `xlabel`, `ylabel`, and `title` do this for us. For instance:

```
>> xlabel('Time (seconds)');
>> ylabel('Amplitude');
>> title('Plot of x[n]');
```

Note that the single tick marks, `'`, delimit *strings* that are passed to these commands.

[5] For Mac users, I believe you double-click to zoom out all the way.

7.4 Commands related to “plot”

There are a few similar commands for plotting vectors as well. Try these commands, and make sure you zoom in on each one so you can see the results:

```
>> stem(x)
>> stairs(x)
>> bar(x)
```

In this course, you will most often be using the `plot` and `stem` commands. Each is useful in a somewhat different context.

7.5 Plotting with an x-axis

When you checked the `help` for `plot` (you *did* look at the `help`, didn't you?), most likely you noticed that there are some more explicit ways to use the function. There is an optional first parameter that gives the x-position of each data point. Thus, we use plot for x-y scatter plots and other things. Calling plot without the first parameter is equivalent to the following command:

```
>> plot(1:length(x),x,'x-');
```

Sometimes, we'll have a *time axis* that we want to plot against. For instance,

```
>> t = 0:.01:1.99;
>> plot(t,x);
```

This scales the time axis to match `t`. We will find this very useful when working with *sampled* signals.

7.6 Plotting multiple vectors on the same figure

It possible (and often desirable) to plot multiple vectors simultaneously. One way (which is probably the easiest to remember) requires a set of parameters for each vector. Execute the following commands:

```
>> y = .8*sin(2*pi*(1:200)/14 + 0.5);
>> plot(t,x,'go-',t,y,'rx--');
```

This plots `x` and `y` versus `t` on the same figure with different line types. Note that the line style arguments are optional; without them, MATLAB will plot each curve using a different color.

The `hold` command provides another method of plotting several curves on the same figure. When we type `hold on`, an old figure will not be erased before a new one is plotted. To add a curve to the plot we produced above, use the commands:

```
>> hold on;
>> plot(t,.3*x,'ks:');
>> hold off;
```

A third way to plot multiple lines simultaneously makes use of the fact that `plot` will plot the columns of a matrix as separate lines. Execute the following commands.

```
>> plot([x; y]');
```


7.7 Legends

You can add a legend to a plot using the `legend` command like this:

```
>> legend('Data set 1', 'Data set 2');
```

The `legend` command can take any number of parameters; usually, though, you want one string for each data set on your plot.

7.8 Putting several axes on one figure

Often we'll want to plot two vectors next to one another but not on the same set of axes. To do this, we use the `subplot` command. `subplot` takes three parameters: the number of rows, the number of columns, and the figure number. Thus, the following command the fourth *subplot* in an array of subplots with three rows and two columns.

```
>> subplot(3,2,4);
```

(Notice that it opens the fourth counting *across the rows*, as you would read a page. This is notably different from single number indexing of MATLAB arrays.)

Now, to put several plots in subplots like this, we simply execute several subplot commands like this:

```
>> subplot(2,1,1);
>> plot(1:10, (1:10).^2);
>> subplot(2,1,2);
>> plot(1:10, (1:10).^3);
```

7.9 Two-dimensional arrays

You're probably not surprised by now that MATLAB also has facilities for visualizing two dimensional arrays. Let's look at some of them. First, we need an interesting matrix to look at. Execute the following command:

```
>> z = membrane(1,50);
```

We now have a 101x101 matrix of numbers. The most straightforward way to look at this data is using the `imagesc` command, which displays the matrix as though it were an image. Execute the following commands:

```
>> imagesc(z); axis xy; colorbar;
```

Our surface has been displayed in color. Notice the colorbar along the right side of the image, which tells what values the various colors map to. This type of display, where different colors are used to represent different values, is known as a pseudocolor display. If we look at the image we've got a "high" spot in the lower right that tapers off to "low" regions around the outside. The surface also has an overall L-shape. Another way to visualize this uses the `contour` command. Try this:

```
>> contour(z,20); colorbar;
```

This display, the *contour plot*, shows us lines of constant height. This is the way that meteorologists usually display atmospheric pressure on weather maps.

We also have some more interesting options. Try each of the following commands separately:


```
>> mesh(z); rotate3d on
>> surf(z); rotate3d on
```

Now we have some "3-D" visualizations of our surface. If you click-and-drag the plot, you should be able to rotate the surface so that you can see it from various directions. Experiment with this until you're comfortable with how it works. Notice what happens if you look at the surface from directly above.

MATLAB has some very powerful tools for data visualization; here, you've seen only a small sampling. There many more. If you're interested in exploring this topic further, check `help graph2d`, `help graph3d`, and `help specgraph`.

8 Programming in MATLAB

Programming in MATLAB is really just like using the MATLAB command line. The only difference is that commands are placed in a file (called an *M-file*) so that they can be executed by simply calling the file's name. We'll also see that MATLAB has many of the same control flow structures, like loops and conditionals, as other, more traditional programming languages.

8.1 Paths and working directories

Before we jump into programming in MATLAB we need to make a few comments about files in MATLAB. MATLAB has access to a machine's file system in roughly the same way a command-line based operating system like DOS or UNIX. It has a "present working directory" (which you can see with the command `pwd`); any files in the present working directory can be seen by MATLAB. You can change the present working directory in roughly the same way that you do in DOS or UNIX, using the `cd` command (for "change directory"). MATLAB also has a "path," like the path in DOS or UNIX, which lists other directories that contain files that MATLAB can see. The `path` command will list the directories in the path. We'll be making a few files in this tutorial, and you'll need to store commands in files when doing the laboratories. You'll probably want to make a directory somewhere in your personal workspace, `cd` to that directory, and store your files there. Unless you're working on your own system, do not store them in the main MATLAB directory; if you do, the system's administrator will probably become very irritated with you.

8.2 Types of command files in MATLAB

There are two types of files containing commands that MATLAB can call, *scripts* and *functions*. Both use the ".m" file extension (and, thus, are called *m-files*. A script is nothing but a list of commands. When you call the script (by simply typing in the script's filename), MATLAB will execute all of the commands in the file and return to the command line exactly as if you had typed the commands in by hand. Functions are different in that they have their own workspace and variables. We pass information to a function by means of input parameters, and receive information from the function through output parameters.

MATLAB scripts

Start the MATLAB editor using the command `edit`[6]. Then, place the following lines in the text file and save it as "hello.m".

```
% hello.m  --  Introductory 'Hello World' script
% These lines are comments, because they start with '%'

hw = 'Hello World!';   % Comments can appear on the same lines
disp(hw);              %   as commands, again after a '%'
```

Now execute it by typing `hello` at the MATLAB prompt. (Remember that the file needs to be in your present working directory or on the path for MATLAB to see it – `cd` to the correct directory if necessary). As a result of executing the script, you should now have a variable 'hw' in your workspace (remember, `who` lists variables in your workspace). Note that scripts make use of (and possibly overwrite) variables in your base workspace. For further information on scripts, type `help script`.

MATLAB functions

The second type of file that we can put commands in is called a *function*. A function communicates with the current workspace by passing variables called *parameters*. It also creates a separate workspace so that it's variables don't get mixed up with whatever variables you have in your current workspace. Note that most MATLAB commands are also functions, and the M-file code is available for most of them. You can see the code by using the `type` command, for instance as `type fliplr`.

Using your text editor, make a new file that contains the following lines and save it as "hello2.m."

```
% hello2.m  --  Introductory 'Hello World' function
% Try typing 'help hello2' when you're done, and see what happens
%
% function output_param = hello2(input_param)

function output_param = hello2(input_param)
% The line above tells MATLAB that this is a function
%   with one input and one output parameter

hw2 = ['Hello World! x' num2str(input_param)];
disp(hw2);
output_param = hw2;
```

To call this function, type `hello2(2)`. Note that once you've done this, the variable `hw2` does not show up in your workspace. However, the data that was stored in `output_param` (the output parameter) has been placed in `ans`. This is exactly what happens if you called a MATLAB built-in function without supplying an output parameter. Similarly, the '2' is an input parameter which is passed into the function. When a function is executed, it will not have any variables defined except those defined inside the function itself and the input parameters. Note that a function does not need to have either input parameters or output parameters. For further help on this, type `help function` at the MATLAB prompt.

[6]While you can use any text editor for editing MATLAB code, the MATLAB editor has a number of useful features for doing so. UNIX versions of MATLAB prior to version 6 did not include a built-in editor.

Scripts versus functions

There are some situations when scripts are more convenient to use, and others where functions are more useful. Scripts are useful for automating some set of commands that you would otherwise need to type into the command line repeatedly. Scripts have full access to your variables, which can be both positive and negative. On the one hand, we do not need to explicitly pass every variable needed to the script. On the other hand, scripts are generally dependent upon the state of the workspace variables. When running scripts, we also risk overwriting variables that we do not wish to overwrite.

Functions, on the other hand, are good for when we wish to perform some task repeatedly with different data. If we want to run a script many times with a different variable setting, we may need to change the variable by hand in the code. With a function, we simply pass that variable into the function as a parameter. Because of their separate workspace, we are guaranteed that a function is only dependent upon the parameters we pass into it. This makes a function more portable from one situation to another (since we don't need to worry about the state of the calling workspace), and generally forces the programmer to be clear about variable initialization and the like. One downside of functions is that it is somewhat harder to see the results of "internal" computations without resorting to debugging (see Section 9).

The writing of functions versus scripts is very much a matter of personal preference. However, we tend to prefer using functions in any situation where doing so is not prohibitively difficult. The encapsulation of data allows for the reuse of functions much more readily than scripts. Perhaps it is telling that nearly all built-in MATLAB commands are functions rather than scripts.

8.3 Control Structures

In MATLAB we also have a number of programming constructs at our disposal. While primarily used in M-files, these constructs can also be used at the command line. However, anything complicated enough to need a loop or an if-statement is usually worth putting into an M-file. Let's look at the most typical types of programming constructs.

Loops

The `for` loop is used to execute a set of commands a certain number of times, while also providing an index variable. Consider the simple loop here:

```
for index = 1:10
   disp(index);
end
```

This loop executes the `disp` command ten times. The first time it is executed, `index` is set to 1. Thereafter, it is incremented by one each time the commands in the loop are executed. Note that the colon form of the `for` loop is not mandatory; any *row-vector* can be used in its place, and the index (which, of course, can be renamed) will be sequentially set to each of the elements in the vector from left to right.

We can use `while` loops in a similar manner. Consider this:

```
ct = 10;
while ct > 0.5
  ct = ct/2;
  disp(ct);
end
```


As long as the conditional after `while` is true, the loop will be executed.

Conditional statements

A more traditional method of conditional execution comes from the `if-else` statement. Consider this:

```
if pi > 4
   disp('Pi is too big!');
elseif pi < 3
   disp('Pi is too small!');
else
   disp('Pi is just about right.');
end
```

Here, MATLAB will first check the conditional, `pi > 4`. If this is true, the first display command will be executed and the remainder of the `if-else` statement will be skipped (that is, none of the other conditionals will be tested). If the first conditional is false, MATLAB will begin to check the remaining conditionals. There can be any number of `elseif` statements in this construct (including none), and the `else` statement is entirely optional. If you have a large number of chained conditionals, you might consider using the `switch-case` construct (type `help switch` or `help case`).

8.4 Strings and string output

In `hello2` above, we constructed a string and displayed it. Though not so useful at the command line, in programming we often want do work with strings and display them. In MATLAB, strings are delimited by the single tick-mark `'`. Thus, `'STRING'` is treated as a literal string, rather than being interpreted as a variable. Strings, though, are just row-vectors of characters. This means that we can build strings using the same vector concatenation operators that we presented earlier. Thus, the following command:

```
>> [ 'string' 'test' ]
```

outputs the string `'stringtest'`.

Rather than echoing strings (or numbers, for that matter) by omitting the semicolon, we can also use the `disp` command. Notice the difference when we call this command:

```
>> disp([ 'string' 'test' ]);
```

Also, for any C programmers in the audience, note that you can perform formatted string output with `fprintf` and `sprintf`.

It is often useful to convert numbers to strings. We can use the `num2str` command to do this. Consider this:

```
>> for counter = 1:10
>>   disp(['Percent completed: ' num2str(10*counter) '%']);
>> end
```

In this way, we can produce formatted output without using `fprintf` or `sprintf`.

For more information on strings, look at `help strings` and `help strfun`.

9 Debugging your MATLAB code

Inevitably, when you put MATLAB commands into a file as a script or a function, you will make mistakes and need to locate them. Because of its interpreted environment, MATLAB is actually one of the most pleasant languages to debug. And, as is always the case when debugging code, there are many ways to accomplish this.

If you are executing a script or function and MATLAB encounters an error, it will immediately print the line number of the function on which the error occurred. If the error occurs in a file other than the calling file, a call stack will be printed. This listing shows which file called which other files and on what line number. This allows us to pinpoint the source of the error quickly.

One of the simplest ways to debug is a method you are probably familiar with from other programming languages. We can force MATLAB to print strings or variables using the `disp` command or by placing the variable name on a line by itself without a semicolon. This way, we can display `for` loop counters or other relevant variables to determine what they contain and exactly when in the program flow the code "breaks."

The real power of MATLAB debugging comes from our ability to "break" at any point in the code and then proceed to execute any MATLAB commands. There are a number of ways to do this. For instance, you can tell MATLAB to stop and enter "debug mode" whenever an error is encountered. When you're in debug mode, the command line changes to `K>>`. You will then have access to all of the variables that are in scope at the time. Turn on this option with the command

```
>> dbstop if error
```

To turn it off again, use the command

```
>> dbclear if error
```

We can also set and clear breakpoints elsewhere in the code using the same commands. To set (and then clear) a breakpoint in `hello2.m` at line 11, call

```
>> dbstop in hello2 at 11
>> dbclear in hello2 at 11
```

`dbstatus` will show all breakpoints that are currently active. Note that if you try to set a breakpoint at a non-command line (such as a comment), the breakpoint will be set at the next valid command.

Another useful command is `dbstep`, which advances one command in the m-file. If you call `dbstep in` or `dbstep out`, you can step into and out of called functions (that is, you traverse up and down the call stack, which contains a list of which functions have been called to reach the current point in the code). `dbstack` lists the current call stack including your current file and the line number in this file. `dbtype` types all or parts of an m-file. Eventually, you'll want to get out of debug mode, so you can call `dbquit` to halt execution of the file or `dbcont` to continue execution until the end of the file or the next breakpoint. In general, `help debug` is the starting point in the help system for learning about the MATLAB command line debugger.

If you are running MATLAB on a Windows system (or possibly a Macintosh), the debugger is also available through the built in editor. The exact implementation depends on your system and the version of MATLAB, but usually breakpoints will show up as red circles next to commands. In debug mode, the current command will be pointed to with an arrow, so you can follow where you are in the code. There are typically shortcut keys and menu items to insert and remove breakpoints, step through the code, and toggle flags such as stop-if-error.

If you save a file that has breakpoints, you may find that your breakpoints disappear. This can be very annoying, so there is an alternative method of entering debug mode. Placing the command

`keyboard` into your code is effectively the same as placing a breakpoint in the code, such that you can execute commands before returning to program execution (with the command `return`).

There are a number of error types that you are likely to encounter. One very good rule of thumb says that if an error occurs inside a MATLAB function, the error is almost assuredly in the calling function. Usually this means that the function is being passed improper parameters; check the call stack or `dbstep` out until you find the line in your program which is causing problems. Other common errors include indexing errors (indexing with 0 or a number greater than the length of the indexed dimension of a variables) and assignment size mismatches. MATLAB is usually pretty descriptive with its error messages once you figure out how to interpret what it is saying. As is usually the case when debugging, an error message at a particular line may in fact indicate an error that has occurred several lines before.

MATLAB reference material

For a useful quick reference for using MATLAB check the end of this laboratory manual starting on page 169. Included are various helpful pieces of information for working with MATLAB.

Laboratory 1

Signals, Signal Statistics, and Signal Detection I

1.1 Introduction

In everyday language, a *signal* is anything that conveys information. We might talk about traffic signals or smoke signals, but the underlying purpose is the same. In the study of *signals and systems engineering*, however, we adopt a somewhat more specific notion of a signal. In this field, a signal is a numerical quantity that varies with respect to one or more independent variables. One can think of a signal as a functional relationship, where the independent variable might be time or position.

As an example, one signal might be the voltage on the wires from a microphone as it varies with time. Another signal might be the light intensity as it varies with position along a sensor array. The important aspect of these signals, though, is the mathematical representation, not the underlying medium. That is, the voltage and light signals might be mathematically the same, despite the fact that the signals come from two very different physical sources. In signals and systems engineering, we recognize that the most important aspects of signals are mathematical. Thus, we don't necessarily need to know anything about the physical behavior of voltage or light to deal with these signals.

What purposes do signals serve? Let us highlight a few of the many important ones. First, a signal can embody a sensory experience, as in a sound that we would like to hear or a picture that we would like to see. Second, a signal can convey symbolic information, as in the text of a newspaper. Third, a signal can serve to control some system. For example, in a typical modern automobile, an electronic control signal determines how much gasoline is emitted by the fuel injectors. Last, we mention that a signal can embody an important measurement, for example, the speed of a vehicle or the EKG of some patient.

What is the advantage of having a sound or a picture or text or control information or a measurement embodied in a signal? For one thing, it enables us to transmit it to a remote location or to record it. In many, but not all, cases, these are done electronically, either with analog or digital hardware. For another, the signals we encounter frequently need to be *processed*, which can also be done electronically with analog or digital hardware. For example, a signal may contain unwanted noise that needs to be removed; this is an example of what is called *noise reduction* or *signal recovery*. Alternatively, the desired information or sensory experience may need to be extracted from the signal, as in the case of AM and FM radio signals, which need to be *demodulated* before we can listen to them, or in the case of CT scan signals, which need extensive processing before an X-ray

like image can be viewed.

Finally, in many situations, the purpose of signals is to permit decisions to be made. This kind of signal processing is variously called *signal classification*, *signal recognition*, or *signal detection*. As examples, a radar system processes the signal received from its antenna to determine whether or not it contains a reflected pulse, which would indicate the presence of an airplane in the direction to which the antenna is pointed. The bill changer in a vending machine processes the signal produced by its optical sensor to determine if the inserted piece of paper is a valid dollar bill. A speech recognition system processes the signal produced by a microphone to determine the words that are spoken. A speaker recognition system processes the same signal to determine the identity of the person speaking. A heart monitoring system processes an electrical EKG signal to determine if arythmia is occuring. A digital modem processes the received signal to determine what bits are being transmitted. These are a just a few of the situations in which signals must be processed to make decisions.

As one of part of this lab assignment, we will implement and tune a simple signal processing system for detecting whether or not a recorded signal contains a spoken sound. In later lab assignments, we will develop more sophisticated decision making systems – for detecting the presence or absence of radar pulses, for decoding a sequence of key presses from the signal produced by a touchtone telephone, and for deciding which of several vowel sounds has been spoken into a microphone.

Throughout this course we will develop tools for analyzing, modifying, processing and extracting information from signals mathematically. One of the most basic (and sometimes most useful) methods involves the calculation of *signal statistics*. Calculating signals statistics provides us a substantial amount of useful information about a signal. These statistics allow us to determine "how much" signal is present (i.e., the *signal strength*), how long a signal lasts, what values the signal takes on, and so on. We will use signal statistics to develop measures of *signal quality* (with respect to a reference signal) and also to perform *signal detection* (by determining when a signal contains useful information rather than just background noise).

1.1.1 "The Questions"

- How can I quantitatively determine a signal's "quality"?
- How can I detect the presence of "speech" within a segment of a speech signal?

1.2 Background

1.2.1 Continuous-time and discrete-time signals

In its most elementary form, a *signal* is a time-varying numerical quantity, for example, the time-varying voltage produced by a microphone. Equivalently, a signal is a numerically valued function of time. That is, it is an assignment of a numerical value to each instance of time. As such, it is customary to use ordinary mathematical function notation. For example, if we use s to denote the signal, i.e. the function, then $s(t)$ denotes the *value* of the signal at time instance t. In common usage, the notation $s(t)$ also has an additional interpretation — it may also refer to the entire signal. Usually, the context will make clear which interpretation is intended.

We will deal with many different signals and to keep them separate we will use a variety of symbols to denote them, such as such as r, x, y, z, x'. Occasionally, we will use other symbols to denote time, such as t', s, u. In some situations, the independent parameter t represents something

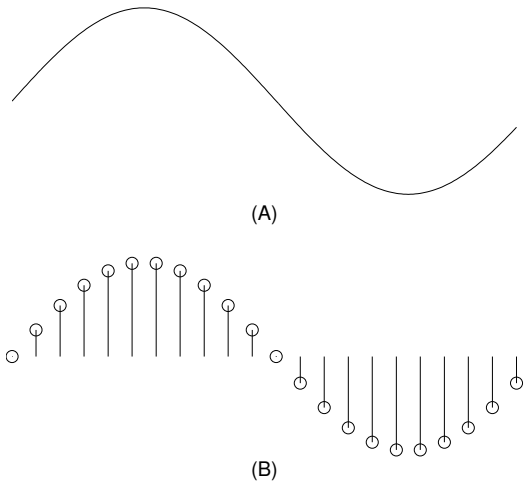


Figure 1.1: (A) A continuous-time signal. (B) A discrete-time signal.

other than "time", such as "distance". This happens, for example, when pictures are considered to be signals.

As illustrated in Figure 1.1, there are two basic kinds of signals. When the time variable t ranges over all real numbers, the signal $s(t)$ is said to be a *continuous-time* signal. When the time variable t ranges only over the set of integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$, the signal $s(t)$ is said to be a *discrete-time* signal. To distinguish these, from now on we will use a somewhat different notation for discrete-time signals. Specifically, we will use one of the letters i, j, k, l, m, n to denote the time variable, and we will enclose the time-variable in square brackets, rather than parentheses, as in $s[n]$. Thus, for example, $s[17]$ denotes the value of the discrete-time signal $s[n]$ at time $n = 17$. Note that for discrete-time signals, the time argument has no "units". For example, $s[17]$ simply indicates the 17th sample of the signal. When the independent parameter t or n represents something other than time, for example distance, then the signal can be said to be *continuous-space* or *discrete-space*, respectively, or more generally, *continuous-parameter* or *discrete-parameter*.

It is important to reemphasize the inherent ambiguity in the notation $s(t)$ and $s[n]$. Sometimes $s(t)$ refers to the value of the signal at the specific time t. At other times, $s(t)$ refers to the entire signal. Usually, the intended meaning will be clear from context. The same two potential interpretations apply to $s[n]$.

1.2.2 Describing Signals

Some continuous-time signals can be described with formulas, such as $s(t) = \sin(t)$ or

$$s(t) = \begin{cases} 0 & t < 0 \\ \cos(t) & t \geq 0 \end{cases} . \tag{1.1}$$

For other signals, there are no such formulas. Rather they might simply be measured and recorded, as with an analog tape recorder. Similarly, some discrete-time signals can be described with formulas, such as $s[n] = \sin(n)$ or

$$s[n] = \begin{cases} 0 & n < 0 \\ \cos[n] & n \geq 0 \end{cases} , \tag{1.2}$$

and some are described simply by recording their values for all values of n.

Often, a discrete-time signal is obtained by *sampling* a continuous-time signal. That is, if T_s is a small time increment, then the discrete-time signal $s[n]$ obtained by sampling $s(t)$ with *sampling interval* or *sampling period* T_s is defined by

$$s[n] = s(nT_s)\ , -\infty < n < \infty \tag{1.3}$$

For example, if $s(t) = \sin(t)$ and $T_s = 3$, then the discrete-time signal obtained by sampling with sampling interval T_s is $s[n] = \sin(3n)$. The reciprocal of T_s is called the *sampling rate* or *sampling frequency* and denoted $f_s = 1/T_s$. Its units are samples per second. The discrete-time signal in Figure 1.1 was obtained by sampling the continuous-time signal shown above it.

In the above example, we have allowed the time parameter to be negative as well as positive, which begs the question of how to interpret negative time. Time 0 is generally taken to be some convenient reference time, and negative times simply refer to times prior to this reference time.

Nowadays, signals are increasingly processed by digital machines such as computers and DSP chips. These are inherently discrete-time machines. They can record and work with a signal in just two ways: as a formula or as a sequence of samples. The former applies to continuous-time and discrete-time signals. For example, a computer can easily compute the value of the continuous-time signal $s(t) = \sin(t)$ at time $t = \sqrt{2}$ or the value of the discrete-time signal $s[n] = \cos(n)$ at time $n = 17$. However, the latter works only with discrete-time signals. Thus, whenever a digital machine works with a continuous-time signal, it must either use a formula or it must work with its samples. That is, it must work with the discrete-time signal that results from sampling the continuous-time signal. This admonition applies to us, because in this and future lab assignments, many of the signals in which we are interested are continuous-time, yet we will process them exclusively with digital machines, i.e. ordinary computers.

Except in certain ideal cases, which never apply perfectly in real-world situations, sampling a continuous-time signal entails a "loss". That is, the samples only partially "capture" the signal. Alternatively, they constitute an approximate representation of the original continuous-time signal. However, as the sampling interval decreases (equivalently, the sampling rate increases), the loss inherent in the sampled signal decreases. Thus in practical situations, when the sampling interval is chosen suitably small, one can reliably work with a continuous-time signal by working with its samples, i.e. with the discrete-time signal obtained by sampling at a sufficiently high rate. This will be the approach we will take in this and future lab assignments, when working with continuous-time signals that cannot be described with formulas.

When digital machines are used to process signals, in addition to sampling, one must also *quantize*, or *round*, the sampled signal values to the limited precision with which numbers are represented in the machine, e.g. to 32-bit floating point. This engenders another "loss" in the signal representation. Fortunately, for the computers we will use in performing our lab experiments, this loss is so small as to be negligible. For example, MATLAB uses 64-bit double-precision floating point representation of numbers. (Lab 5 is an exception; in that lab, we will consider systems that are designed to produce digital signal representations with as few bits as possible.)

1.2.3 Signal support and duration

The *support* of a signal is the smallest time interval that includes all non-zero values of the signal. For example, the continuous-time signal $s(t) = \cos(t), 0 \leq t \leq 3, s(t) = 0$, else has support interval $[0, 3]$. The discrete-time signal $s[n] = \cos(n), 0 \leq n \leq 3$, has support interval $[0, 3] = \{0, 1, 2, 3\}$. The *duration* of a signal is simply the length of its support interval. In the previous

examples, the duration of $s(t)$ is 3, and the duration of $s[n]$ is 4. Note that the support and duration of a signal can be either finite or infinite.

1.2.4 Periodicity

Periodicity is a property of many naturally occurring or man-made signals. A continuous-time signal $s(t)$ is said to be periodic with period T, where T is some positive real number, if

$$s(t+T) = s(t) \text{ , for all } t \tag{1.4}$$

If $s(t)$ is *periodic with period* T, then it is also periodic with period $2T$, $3T$, ... The *fundamental period* T_o of $s(t)$ is the smallest T such that $s(t)$ is periodic with period T.

Similarly, a discrete-time signal $s[n]$ is said to be periodic with period N, where N is some positive integer, if

$$s[n+N] = s[n] \text{ , for all } n \tag{1.5}$$

If $s[n]$ is periodic with period N, then it is also periodic with period $2N$, $3N$, ... The fundamental period N_o is the smallest N such that $s[n]$ is periodic with period N.

1.2.5 Signals in MATLAB

While signals can be represented by formulas or by recorded signal values, when working in MATLAB, we generally use the latter. That is, we represent a signal as a *vector* (i.e., a one-dimensional *array*) of numbers.

Discrete-time signals

We begin with an example. Suppose we want to represent the following discrete-time signal as an array in MATLAB:

$$s[n] = \begin{cases} n^2 & 5 \le n \le 15 \\ 0 & \text{else} \end{cases} \tag{1.6}$$

In MATLAB, we do this by creating two vector: a *support vector* and a *value* or *signal vector*. The support vector represents the *support interval* of the signal, i.e. the set of integers from the first time at which the signal is nonzero to the last. For this example, the support vector can be created with the command

```
>> n = 5:15
```

This causes `n` to be the array of 11 numbers $5, 6, \ldots, 15$. Next, the signal vector can be created with the command

```
>> s = n.^2
```

which causes `s` to be the array of 11 numbers $25, 36, \ldots, 225$.

Note that as in the above example, we usually only specify the signal within the range of times that it is nonzero. That is, we usually do not include zero values outside the support interval in the signal vector.

It is often quite instructive to plot signals. To plot the discrete-time signal `s[n]`, use the stem command:

```
>> stem(n,s)
```


You can also use the `plot` command; however, `plot` draws straight lines between plotted points, which may not be desirable.

It is important to note that in MATLAB, when `i` is an integer then `s(i)` is not necessarily the signal value at time `i`. Rather it is the signal at time `n(i)`. Thus, `stem(n,s)` and `stem(s)` result in similar plots with different labelings of the time axis. Occasionally, it will happen that `n(i) = i`, in which case `s(i) = s(n(i))` and `stem(n,s)` and `stem(s)` result in identical plots with identical time axis labels.

Continuous-time signals

We begin with an example. Suppose we wish to represent the following continuous-time signal as an array in MATLAB:

$$s(t) = \begin{cases} t^2 & 5 \leq t \leq 15 \\ 0 & \text{else} \end{cases} \tag{1.7}$$

We first choose a sampling interval `Ts` with a command such as

```
>> Ts = 1/20
```

We then create a support vector with the command

```
>> t = 5:Ts:15
```

Finally, we create a signal vector with the command

```
>> s = t.^2
```

What have we done? To represent $s(t)$, we have created a support vector `t` that contains the sample times $5, 5 + 1/20, 5 + 2/20, 5 + 3/20, \ldots, 15$, and we have created a signal vector `s` that contains the samples $s(5), s(5 + 1/20), s(5 + 2/20), s(5 + 3/20), \ldots, s(15)$. That is, for $n = 1, \ldots, 301$, `s(n)` contains the signal value at time `t(n) = 5 + (n-1)/20`.

Note that when representing a continuous-time signal as an array, it is usually important to choose the sampling interval `Ts` small enough that the signal changes little over any time interval of `Ts` seconds.

As with discrete-time signals, it is frequently instructive to plot a continuous-time signal. This is done with the command

```
>> plot(t,s)
```

which plots the points `(t(1),s(1))`, `(t(2),s(2))`, ..., and connects them with straight lines. Connecting these points in this manner produces a plot that approximates the original continuous-time signal $s(t)$, which takes values at all times (not just integers) and which usually does not change significantly between samples (assuming `Ts` is chosen to be small enough). Note that `plot(s)` produces a similar plot, but the horizontal axis is labeled with sample "indices" (i.e., the number of the corresponding sample) rather than sample times. When working with continuous-time signals, it is important that you always use `plot(t,s)` rather than `plot(s)`. It also important that your plot indicates what the axes represent, which can be done using the `xlabel` and `ylabel` commands.

1.2.6 Signal Statistics

When dealing with a signal, it is often useful to obtain a rough sense of the range of values it takes and of the average size of its values. We do this by computing one or more *signal statistics*.

The following lists a number of common signal statistics. It gives the defining formula for each for both continuous-time and discrete-time signals. Also included is MATLAB code for calculating the statistic[1] for a discrete-time signal. If we wish to compute a statistic for a continuous-time signal when we only have a sampled representation, we can use the discrete-time statistic to approximate the continuous statistic. The formulas needed for this approximation are included here with the label "sampled." (In most cases, this approximation becomes better as the sampling interval T_s decreases.) For completeness, signal support and duration are also defined below.

1. **Support Interval**. A signal's *support interval* (also occasionally known as just the signal's *support* or its *interval*) is the smallest interval that includes all non-zero values of the signal.

$$\text{Continuous-time:} \quad t_1 \leq t \leq t_2 \tag{1.8}$$
$$\text{Discrete-time:} \quad n_1 \leq n \leq n_2 \tag{1.9}$$
$$\text{MATLAB:} \quad \texttt{n = n1:n2} \text{ for a signal } \texttt{s}. \tag{1.10}$$

2. **Duration**. The *duration* of a signal is simply the length of the support interval.

$$\text{Continuous-time:} \quad t_2 - t_1 \tag{1.11}$$
$$\text{Discrete-time:} \quad n_2 - n_1 + 1 \tag{1.12}$$
$$\text{MATLAB:} \quad \text{Assumed } \texttt{length(s)} \text{ for a signal } \texttt{s}. \tag{1.13}$$
$$\text{Sampled:} \quad (t_2 - t_1) = (n_2 - n_1 + 1)T_s \tag{1.14}$$

3. **Periodicity**. Periodicity was described in section 1.2.4. The key formulas are included here.

$$\text{Continuous-time:} \quad s(t) = s(t + T) \tag{1.15}$$
$$\text{Discrete-time:} \quad s[n] = s[n + N] \tag{1.16}$$
$$\text{Sampled:} \quad T \approx NT_s \tag{1.17}$$

4. **Maximum and Minimum Value**. These values are the largest and smallest values that a signal takes on over some interval defined by n_1 and n_2. In MATLAB these values are found using the `min` and `max` commands.

$$\text{MATLAB:} \quad Maximum(\texttt{s}) = \texttt{max(s)} \tag{1.18}$$
$$\text{MATLAB:} \quad Minimum(\texttt{s}) = \texttt{min(s)} \tag{1.19}$$

5. **Average Value**. The *average value*, M, is the value around which the signal is "centered"

[1] This code assumes that the signal vector `s` is defined only over the range of times for which we wish to compute the statistic. More generally, if `n` is the support vector and `n1` and `n2` define a subset of the support vector over which we wish to calculate our statistic, we can compute the statistic over only this range, `n1:n2`, by replacing the signal `s` with the shorter signal `s((n1:n2)-n(1)+1)`.

over some interval.

$$\text{Continuous-time:} \quad M(s(t)) = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} s(t)dt \tag{1.20}$$

$$\text{Discrete-time:} \quad M(s[n]) = \frac{1}{n_2 - n_1 + 1} \sum_{n=n_1}^{n_2} s[n] \tag{1.21}$$

$$\text{MATLAB:} \quad M(\texttt{s}) = \texttt{mean(s)} \tag{1.22}$$

$$\text{Sampled:} \quad M(s(t)) \approx M(s[n]) \tag{1.23}$$

6. **Mean-squared value**. The *mean-squared value* (or *MSV*) of a signal, MS, is defined as the average squared valued of the signal over an interval. The MSV is also called the **average power**, because the squared value of a signal is considered to be the instantaneous power of the signal.

$$\text{Continuous-time:} \quad MS(s(t)) = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} s^2(t)dt \tag{1.24}$$

$$\text{Discrete-time:} \quad MS(s[n]) = \frac{1}{n_2 - n_1 + 1} \sum_{n=n_1}^{n_2} s^2[n] \tag{1.25}$$

$$\text{MATLAB:} \quad MS(\texttt{s}) = \texttt{mean(s.\^{}2)} \tag{1.26}$$

$$\text{Sampled:} \quad MS(s(t)) \approx MS(s[n]) \tag{1.27}$$

7. **Root mean squared value**. The *root mean squared value* (or *RMS value*) of a signal over some interval is simply the square root of mean squared value.

$$\text{Continuous-time:} \quad RMS(s(t)) = \sqrt{\frac{1}{t_2 - t_1} \int_{t_1}^{t_2} s^2(t)dt} \tag{1.28}$$

$$\text{Discrete-time:} \quad RMS(s[n]) = \sqrt{\frac{1}{n_2 - n_1 + 1} \sum_{n=n_1}^{n_2} s^2[n]} \tag{1.29}$$

$$\text{MATLAB:} \quad RMS(\texttt{s}) = \texttt{sqrt(mean(s.\^{}2))} \tag{1.30}$$

$$\text{Sampled:} \quad RMS(s(t)) \approx RMS(s[n]) \tag{1.31}$$

8. **Signal Energy**. The *energy* of a signal, E, indicates the strength of a signal is present over some interval. Note that energy equals the average power times the length of the interval.

$$\text{Continuous-time:} \quad E(s(t)) = \int_{t_1}^{t_2} s^2(t)dt \tag{1.32}$$

$$\text{Discrete-time:} \quad E(s[n]) = \sum_{n=n_1}^{n_2} s^2[n] \tag{1.33}$$

$$\text{MATLAB:} \quad E(\texttt{s}) = \texttt{sum(s.\^{}2)} \tag{1.34}$$

$$\text{Sampled:} \quad E(s(t)) \approx E(s[n])T_s \tag{1.35}$$

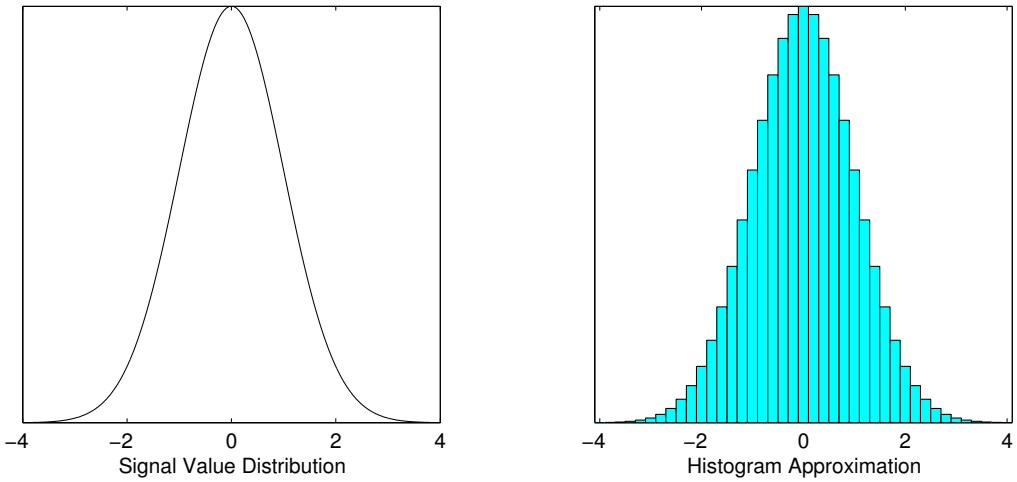


Figure 1.2: Signal value distribution and a discrete histogram approximation

9. **Signal Value Distribution**. The *signal value distribution* is a plot indicating the relative frequency of occurrence of values in a signal. There is no closed-form definition of the signal value distribution, but it can be approximated using a *histogram*. A histogram counts the number of samples that fall within particular ranges, or *bins*. Note that the y-axis is effectively arbitrary, and that the coarseness of the approximation is determined by the number of histogram bins that are used. Figure 1.2 shows an example of a signal value distribution and the histogram approximation to that distribution.

MATLAB: `hist(s,num_bins);` (1.36)

1.2.7 Measuring signal distortion and error

Suppose that we wish to transmit a signal from one location to another. This is a common task in *communication systems*. A common problem is that the signal is often modified or *distorted* in the communication process. Thus, the received signal is not the same as the transmitted signal. Typically, we want to reduce the amount of distortion as much as possible. However, this requires that we have a method of measuring the amount of distortion in a signal. In order to develop such a measure, we'll look at a *signal plus noise* model of signal distortion.

Suppose we are transmitting a signal $s[n]$ over FM radio. Someone tunes in to our radio station and receives a modified version of our signal, $r[n]$. We can represent this modification mathematically as the addition of an *error signal*, $v[n]$, like this:

$$r[n] = s[n] + v[n]. \tag{1.37}$$

Assuming that we have both $s[n]$ and $r[n]$, we can easily calculate $v[n]$ as

$$v[n] = r[n] - s[n]. \tag{1.38}$$

Note that if $s[n]$ and $r[n]$ are identical, $v[n]$ will be zero for all n. This suggests that we can simply measure the signal strength of $v[n]$ by using one of the energy or power statistics.

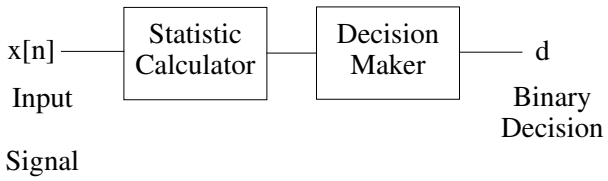


Figure 1.3: An "overview" block diagram for a "signal present" detector.

Mean squared value is a natural choice because it normalizes the error with respect to the length of the signal. Sometimes, though, the RMS value is more desirable because it produces error values that are directly comparable to the values in $v[n]$[2]. When we measure the MSV of an error signal, we sometimes call it the *mean squared error* or *MSE*. Similarly, the RMS value of an error signal is often called the *root mean squared error* or *RMSE*.

In MATLAB, we will usually want to calculate the MSE or RMSE over the entire length of the signals that we have. Supposing that we are given a signal `s` and a modified version `s_mod` (with the same size), we can calculate the MSE and RMSE like this:

```
>> mse = mean((s - s_mod).^2);
>> rmse = sqrt(mean((s - s_mod).^2));
```

Notice that we could also subtract `s` from `s_mod`; the order doesn't matter because of the square operation. Also note that you *must* include the period before the exponentiation operator in order to correctly square each sample.

1.2.8 Signal detection

Suppose that we are designing a continuous speech recognition and transcription system for a personal computer. The computer has a microphone attached to it, and it "listens" to the user's speech and tries to produce the text that was spoken. However, the user is not speaking continuously; there are periods of silence between utterances. We don't want to try to recognize speech where there is silence, so we need some means of determining when there is a speech signal present.

This is an example of *signal detection*. There are many different types of signal detection. Sometimes signal detection involves finding a signal that is obscured by noise, such as radar detection. In other applications, we need to determine if a particular signal exists in a signal that is the sum of many signals. The "signal present" detector for our speech recognition system is a simpler form of signal detection, but it still important in many applications. As another example, some digital transmission systems send bits using what is known as *on/off keying*. They send an electrical pulse to represent "1" and send nothing at all to represent "0". The receiver for such a system uses a "pulse present" detector.

In this laboratory, we will consider the design of a "signal present" detector to identify spoken segments of a speech signal. Note, however, that the detector we will design can be used for many other applications as well. Figure 1.3 shows a block diagram of such a detector. The detector consists of two blocks. The first block computes one or more statistics that we will use to perform the detection. The second block uses the computed statistic(s) to make the final decision.

[2]Mean square values are comparable to the square of the values in $v[n]$

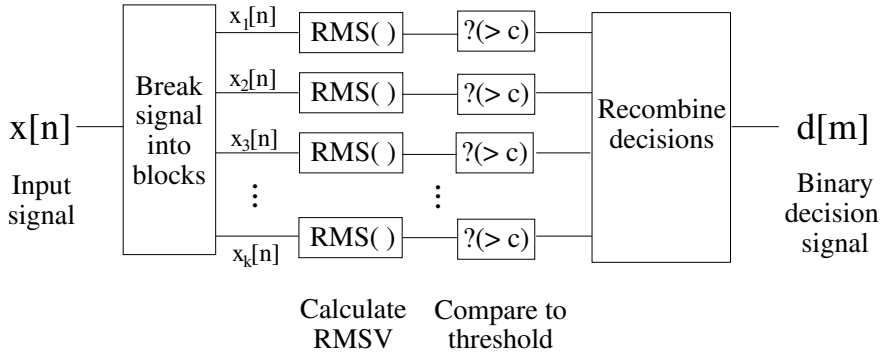


Figure 1.4: A detailed block diagram for the "signal present" detector.

Specifying the detector's operation

The first step is to specify what our system needs to do. From the description above, we know that we will receive a signal as an input. For simplicity, we'll assume that we are given an entire discrete-time signal. What must our system do? We need some sort of indication as to when speech is present in a signal. However, the signal that we are given will contain both periods of silence and periods with speech. For some integer N, let us break the signal into *blocks* of N samples, such that the first N samples make up the first block, the next N samples make up the second block, and so on. Then, we will make a "speech present" or "speech not present" decision separately for each block. The output of our system will consist of a signal with a 0 or 1 for each block, where a 1 denotes "speech present" and a 0 denotes "speech not present". To describe this signal in MATLAB, our system will produce a signal support vector containing containing the index of the first sample of each block and a signal vector containing the 0 or 1 for each block. Choosing the support vector in this way will allow us to plot the decisions on the same figure as the signal itself.

How will we make the decision for each block? Since we can assume a signal that is relatively free of noise, we can simply calculate the energy for each block and compare the result to a threshold. If the statistic exceeds the threshold, we decide that speech is present. Otherwise, we decide that speech is not present. Using signal energy, though, is not ideal; the necessary threshold will depend on the block size. We may want to change the block size, and we should be able to keep the threshold constant. Using average power is a better option, but we would like our threshold to have a value comparable to the values of the signal. Thus, the RMS value seems to be an ideal choice, and this is what we will use. In summary, for each block the detector computes the RMS value R and compares it to a threshold c. The decision is

$$\begin{aligned} \text{signal present}, &\quad \text{if } R \geq c \\ \text{signal not present}, &\quad \text{if } R < c \end{aligned}$$

Note that our detector system will has two design parameters. One is the block size N, and the other is the threshold c. To tune the system, we will need to find reasonable values for these parameters when we make the detector itself. A more detailed block diagram of the detector can be found in Figure 1.4.

Detector algorithm

Now that we've specified the behavior of the detector, let's come up with an algorithm for performing the detection. Of course, this is not the only way to implement this detector.

It is assumed that the input signal is contained in a signal vector `x` whose support vector is simply `1,2,...,length(x)`.

- Define a variable called `block_size`, representing N.
- Define a variable called `threshold`, representing c.
- Calculate the `number_of_blocks`.
- Determine the support vector `support_output` for the output signal.
- For each block of the signal:
  - Calculate the RMS value of the current block
  - Compare the RMS value to the threshold
  - Store the result in the `output` array
- Return the `output` array

What follows are some details about the algorithm:

1. First, note that we want `number_of_blocks` to be an integer. For this calculation, recall that the function `length` returns the number of samples in a vector. Also, note that the `floor` command rounds down to the nearest integer
2. Suppose that the block size is 512. Then the vector `support_output` should contain the numbers `[1, 513, 1025, ...]` and so on. You can generate `support_output` with a single line of code by using the `:` operator[3].
3. There are actually two separate ways to implement the "for each block" part of this algorithm in MATLAB. One involves using a `for` loop, while the other makes use of the `reshape` command and MATLAB's vector and matrix arithmetic capabilities. Both take roughly the same amount of code, but the second way is somewhat faster. You can implement whichever version of the algorithm that you choose in the lab assignment.
   (a) If you implement the algorithm using a `for` loop, you should first initialize the `output` array to "empty" using the command "`output =[];`". Then, loop over the values in `support_output`. Within the loop, you need to determine what values of n_1 and n_2 to use in the RMS value calculation for a given value of the loop counter. Then, compare the RMS value that you calculate to the threshold and append the result to the end of `output`[4].
   (b) An alternative to the `for` loop is to use the `reshape` command to make a matrix out of our signal with one block of the signal per column. If you choose to use `reshape`, you first need to discard all samples beyond the first `block_size` × `number_of_blocks` samples of the input signal. `reshape` this shorter signal into a matrix with `block_size`

[3] Type `help colon` if you need assistance with this operator.
[4] Use either `output(end+1) = result;` or `output = [output, result];`

rows and `number_of_blocks` columns[5]. Then, use `.^` to square each element in the matrix, use `mean` to take the mean value of each column, and take the `sqrt` of the resulting vector to produce a vector of RMS values. Finally, compare this vector to `threshold` to yield your output vector.

1.3 Some MATLAB commands for this lab

- **Zooming on Figures:** In MATLAB, you can interactively zoom in and out on figure windows. To do so, you can either find a "+ magnifying glass" icon on the figure window, or you can type `zoom on` at the command lines. Then, you can click and drag a zoom box on the figure window to get a closer look at that portion of the figure. Also very useful is the `zoom xon` command, which enables zooming *only* in the x-direction; this is usually how we will want to zoom in on our signals.

- **Using line styles and `legend`:** Whenever you plot two or more signals on the same set of axes, you must make sure that the signals are distinguishable and labeled. Generally, we do this using line styles and the `legend command`. The `plot` command gives you a wide range of options for changing line styles and colors. For instance, the commands

```
>> hold on
>> plot(1:10,1:10,'-')
>> plot(1:10,2:11,':')
>> plot(1:10,3:12,'--')
```

  plot lines using solid, dotted, and dashed lines. Type `help plot` for more details about using different line styles and colors. The `legend` command adds a figure legend for labeling the different signals. For instance, the command

```
>> legend('Solid (lower)', 'Dotted (middle)', 'Dashed (higher)')
```

  adds a legend with labels for each of the three signals on the figure. Note that signal labels are given in the order that the signals were added to the figure.

- **Labeling Figures:** Any time that you create a figure for this laboratory, you need to include axis labels, a figure number, and a caption:

```
>> xlabel('This is the x-axis label');
>> ylabel('This is the y-axis label');
>> title('Figure 1: This is a caption describing the figure');
```

  Note that it is recommended that you use your word processor to produce figure numbers and captions, rather than using the `title` command. You also need to include the code that you used to produce the figures, including label commands. Note that each `subplot` of a figure must include its own axis labels.

- **Function Headers:** At the top of the file containing a function declaration, you must have a line like this:

[5] Remember to assign the output of `reshape` to something! No MATLAB function ever modifies its input parameters unless you explicitly reassign the output to the input.


```
function [out1, out2, out3] = function_name(in1, in2, in3)
```

where `in1`, `in2`, and `in3` are input parameters and `out1`, `out2`, and `out3`. Note that you can name the parameters anything you like, and there can be any number of them. The word `function` is a MATLAB keyword. Also, you do not need to explicitly return the output parameters. Instead, MATLAB will take their values at the end of the function's execution and return them to the calling function.

- **`sum`, `mean`, `min`, and `max`:** Given a vector (i.e., a one-dimensional array), these MATLAB functions calculate the sum, mean, minimum, and maximum (respectively) of the numbers in the array and returns a single number. If these functions are given a matrix (i.e., a two-dimensional array), they calculate the appropriate statistic on each column of the matrix and return a row-vector containing one result for each column.

- **`for` loops:** Most `for` loops in MATLAB have the following form

```
>> for index = 1:20
>>   % This loop is executed twenty times
>> end
```

In this case, the loop counter, `index`, is set to 1 on the first execution loop, 2 on the second, and so on. The loop will execute a total of 20 times. We can, of course, use any variable for the loop counter.

A more general form of the `for` loop is given by

```
>> for index = row_vector
>>   % Code in the loop goes here
>> end
```

where `index` is the loop counter and `row_vector` is a row vector that contains all the values that will be assigned to `index` on successive iterations of the loop. Thus, the loop will execute `size(row_vector,2)` times[6].

- **Reshaping arrays:** The `reshape` command is used to change the shape of an array:

```
>> new_array = reshape(array,[new_rows, new_columns]);
```

`array` must have `new_rows*new_columns` elements[7]. `new_array` will have dimensions of `new_rows` × `new_columns`.

- **Logical operators:** The logical operators (>, <, >=, <=, ==, ~=) perform a test for equality or various forms of inequality. They all operate in the same manner by evaluating to 1 ("true") or 0 ("false") depending upon the truth value of the operator. After executing the following statement, for instance,

```
>> result = (x > 0);
```

`result` will contain a one or a zero if `x` is a single number. If `x` is an array, `result` will be an array of ones and zeros with a size equal to `x`, where each element indicates whether the corresponding element in `x` is in fact greater than 0.

[6] If `row_vector` is actually a matrix (or a column vector), each column of `row_vector` will be assigned to `index` in turn.

[7] The number of elements in `array` can be checked using the command `prod(size(array))`.

1.4 Demonstrations in the Lab Session

- Laboratory policies
- Signals in MATLAB – sampling
- Signal statistics
- Approximating continuous-time signal statistics with sampled signals
- Model of discrete-time signal as signal plus noise
- The "signal present" detector

1.5 Laboratory Assignment

Note that in this and all following laboratory assignments, the bullets (i.e, •) indicate items that you must include in your laboratory.

1. (A simple signal and its statistics) Use the following MATLAB commands to create a signal:

```
>> n = 1:50;
>> s = sin(2*pi*n/50);
```

   (a) (Plotting a signal with labels) Use `stem` to plot the signal. Make sure that you include[8]:
   - The figure itself[9].
   - An x-axis label and a y-axis label.
   - A figure number and a caption that describes the figure.
   - The code you used to produce the signal and the figure. This should be included in an appendix at the end of your report[10]. Make sure you clearly indicate which problem the code belongs to.

   (b) (Calculating signal statistics) Calculate the following statistics over the length of the signal (i.e., let $n_1 = 1$ and n_2 = `length(s)`), and include your results in your report[11].
   - Maximum value
   - Minimum value
   - Mean value
   - Mean squared value
   - RMS value
   - Energy

[8] Note that *every* figure that you produce in a laboratory for this class must include these things!

[9] On Windows systems, you can select "Copy Figure" from Edit menu on the figure window to copy the figure to the clipboard and then paste it into your report. Also, to make your report compact, you should make all figures as small as possible, while being just large enough that the important features are clearly discernable.There are two ways to shrink plots, you can shrink them in your lab report document, or you can shrink the MATLAB window before copying and pasting. Shrinking the MATLAB window is generally preferable because it does not shrink the axis labels. Note, you may need to specify the appropriate copy option, so that what is in fact copied is the shrunk rather than original version of the plot

[10] You should include *all* MATLAB code that you use in the appendix. However, you do not need to include code that is built into MATLAB or code that we provide to you.

[11] Remember to include the code you used to calculate these in your MATLAB appendix.

(c) (Approximating continuous-time statistics in discrete-time) Suppose that `s` is the result of sampling a continuous-time signal with a sampling interval $T_s = 1/100$. Use the discrete-time statistics to estimate the following statistics for the continuous-time signal:

- Signal duration
- Energy
- Average power
- RMS Value

2. (Statistics of real-world signals) Download the file `lab1_data.mat` from the course web page. Place it in the present working directory or in a directory on the path, and type

```
>> load lab1_data
```

This file contains two signals which will be loaded into your workspace. You will use the signal `clarinet`[12] in this problem and in Problem 3. The other signal, `mboc`, will be used in Problem 4.

(a) (Plot the real-world signal) Define the support vector for `clarinet` as `1:length(clarinet)`. Then, use `plot` to plot the signal `clarinet`.

- Include the figure (with axis labels, figure number, caption, and MATLAB code) in your report.

(b) (Zoom in on the signal) Zoom in on the signal so that you can see four or five "periods[13] ."

- Include the zoomed-in figure (with axis labels, figure number, caption, and MATLAB code) in your report.

(c) (Find the signal's period) Estimate the "fundamental period" of `clarinet`. Include:

- Your estimate for the discrete-time signal (in samples).
- Your estimate for the original continuous-time signal (in seconds).

(d) (Approximate the SVD) Use the `hist` command to estimate the signal value distribution of `clarinet`. Use 50 bins.

- Include the figure (with axis labels, code, etc.) in your report.
- From the histogram, make an educated guess of the MSV and RMSV. Explain how you arrived at these guesses.

(e) (Calculate statistics) Calculate the following (discrete-time) statistics over the length of the signal:

- Mean value
- Energy
- Mean squared value
- RMS value

[12] This is a one-second recording of a clarinet, recorded at a sampling frequency of 22,050 Hz. To listen to the sound, use the command `soundsc(clarinet,22050)`.

[13] This signal, like all real-world signals, is not exactly periodic; however, it is approximately periodic.

3. (Looking at and measuring signal distortion) In this problem, we'll measure the amount of distortion introduced to a signal by two "systems." Download the two files `lab1_sys1.m` and `lab1_sys2.m`. Apply each system to the variable `clarinet` using the following commands:

```
>> sys1_out = lab1_sys1(clarinet);
>> sys2_out = lab1_sys2(clarinet);
```

   (a) (Examine the effects of the systems) Use `plot`[14] and MATLAB's zoom capabilities to display roughly one "period" of:
   - The input and output of `lab1_sys1` on the same figure.
   - The input and output of `lab1_sys2` on the same figure.

   (b) (Describe the effects of the systems) What happens to the signal when it is passed through these two systems? Look at your plots from the previous section and describe the effect of:
   - `lab1_sys1.m` on `clarinet`.
   - `lab1_sys2.m` on `clarinet`.

   (c) (Measure the distortion) Calculate the RMS error introduced by each system.
   - RMS error introduced by `lab1_sys1`.
   - RMS error introduced by `lab1_sys2`.
   - Which system introduces the least error? Is this what you would have expected from your plots?

4. (Developing an energy detector) In this problem, we will develop a detector that identifies segments of a speech signal in which speech is actually present. Download the files `sig_nosig.m` and `lab1_data.mat` (if you haven't already) from the course web page. The first is a "skeleton" m-file for the signal/no signal detector function. The second contains a speech signal, `mboc`[15], that we will use to test the detector.

   (a) (Write the detector function) Following the detector description given in Section 1.2.8, complete the function in `sig_nosig.m`. Use a threshold of 0.2 and a block size of 512. Verify the operation of your completed function on the `mboc` signal by comparing its output to that of `sig_nosig_demo.dll`[16].
   - Include the code for your completed version of `sig_nosig.m` in the appendix of your lab report.

   (b) (Plot the results of your function) Call `sig_nosig`[17] like this:

```
>> [detection,n] = sig_nosig(mboc);
```

   Then, plot the output of `sig_nosig` (using "`stairs(n,detection,'k:');`") and the signal `mboc` (with `plot`) on the same figure.

[14] Make sure the two signals are easily distinguishable by using different line styles. Also, any time that you plot multiple signals on a single set of axes, you *must* use `legend` or some other means to label the signals!

[15] This signal has also been recorded with a sampling frequency of 22,050 Hz.

[16] `sig_nosig_demo.dll` is a completed version of the function in `sig_nosig.m`. This demo function has been *compiled* for use on Matlab version 6 on Windows-based systems ONLY.

[17] If you did not successfully complete this function, you may use the compiled demo function for this part of the assignment.

- Include this plot in your report.

(c) (Adjusting the threshold) The threshold given above isn't very good; it causes the detector to miss significant portions of the signal. Change the threshold until all significantly visible portions of the signal are properly marked as "speech" by the detector, but the regions between these portions are marked as "no speech." (Note that you *cannot* use the compiled demo function for this part of the assignment.)

- What threshold did you find?
- Include the figure (like the one you generated in Problem 4b) that displays the output of your detector with this new threshold.

5. On the front page of your report, please provide an estimate of the average amount of time spent outside of lab by each member of the group.

Laboratory 2

Signal Correlation and Detection II

2.1 Introduction

In Lab 1, we designed an energy-based signal/no-signal detector for determining when a desired signal is present. This type of detector has a wide variety of applications, from speech analysis to communication, but it has two weaknesses. First, an energy-based detector is very susceptible to noise, especially when the signal's energy is small compared to the energy of the noise. Second, such a detector cannot distinguish between different types of signals that are mixed together.

In this laboratory, we will examine an alternative detection method that addresses these concerns. It uses a computation called *correlation* to detect the presence of a signal *with a known form*. In general, correlation measures the similarity between two signals. Using correlation for detection has significant applications. For instance, it allows several signals to be sent over a single communications channel simultaneously. It also allows the use of radar and sonar in noisy environments. Later in this course, we will see that correlation forms the basis for one of the most important tools in signals and systems engineering, the *spectrum*.

2.1.1 "The Questions"

- How can we transmit and receive bits from several different users on the same communication channel?

- How can we develop a radar detection scheme that is robust to noise, and how do we characterize its performance?

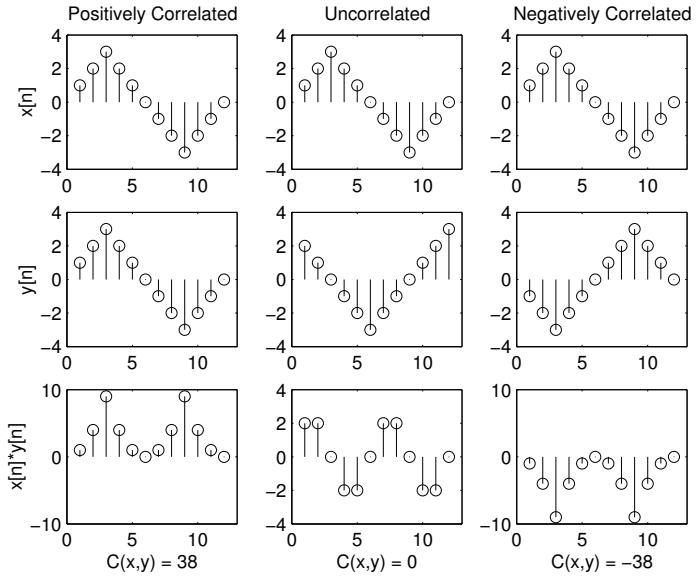


Figure 2.1: Examples of positively correlated, uncorrelated, and anticorrelated signals.

2.2 Background

2.2.1 Correlation

Suppose that we have two discrete-time signals, $x[n]$ and $y[n]$. We compute the *correlation*[1] between these two signals, $C(x, y)$, using the formula

$$C(x, y) = \sum_{n=n_1}^{n_2} x[n]y[n] \tag{2.1}$$

where n_1 and n_2 define the interval over which we are calculating the correlation. In words, we compute a correlation by multiplying two signals together and then summing the product. The result is a single number that indicates the similarity between the signals $x[n]$ and $y[n]$.

What values can $C(x, y)$ take on, and what does this tell us about the signals $x[n]$ and $y[n]$? Let us consider the examples in Figure 2.1. For the signals shown in the first column, $C(x, y) > 0$, in which case, the signals are said to be *positively correlated*. Basically, this means that the signals are more similar than they are dissimilar. In the second column, we can see an example where $C(x, y)$ is zero. In this case, the two signals are *uncorrelated*. One might say that uncorrelated signals are "equally" similar and dissimilar. Notice, for instance, that the signal $x[n] \times y[n]$ is positive as often as it is negative. Knowledge of the value of signal $x[n]$ at time n indicates little about the value of $y[n]$ at time n. Finally, in the third column we see an example where $C(x, y) < 0$, which means that $x[n]$ and $y[n]$ are *negatively correlated*. This means the signals are mostly dissimilar.

Note that the positively correlated signals given in Figure 2.1 are actually identical. This is a special case; from equation (2.1), we can see that in this case the correlation is simply the energy of

[1] We will occasionally refer to this operation as "in-place" correlation to distinguish it from "running" correlation. Sometimes this is also called an "inner product" or "dot product."

$x[n]$, i.e.

$$C(x,x) = E(x)\ . \tag{2.2}$$

Sometimes, it is more useful to work with *normalized correlation*, as defined by

$$C_N(x,y) = \frac{C(x,y)}{\sqrt{E(x)E(y)}} = \frac{1}{\sqrt{E(x)E(y)}} \sum_{n=n_1}^{n_2} x[n]y[n]. \tag{2.3}$$

Normalized correlation is somewhat easier to interpret. The well known Cauchy-Schwartz inequality shows that the normalized correlation varies between -1 and +1. That is, for any two signals

$$-1 \leq C_N(x,y) \leq 1. \tag{2.4}$$

Thus, signals that are as positively correlated as possible have normalized correlation 1 and signals that are as negatively correlated as possible have normalized correlation -1. Moreover, it is known that two signals have normalized correlation equal to 1 when and only when one of the signals is simply the other multiplied by a positive number. In this case, the signals are said to be *perfectly correlated*. Similarly, two signals have normalized correlation equal to -1 when and only when one is simply the other multiplied by a negative number, in which case they are said to be *perfectly anticorrelated*.

2.2.2 Running correlation

In many situations, it is quite useful to correlate a signal $y[n]$ with a sequence of delayed versions of another signal $x[n]$. That is, we wish to correlate $y[n]$ with $x[n]$, with $x[n-1]$, with $x[n-2]$, etc. In such cases, we perform *running correlation* of $y[n]$ with $x[n]$, which produces the *correlation signal*

$$\begin{aligned} r[k] &= C(x[n-k], y[n])\ , k = 0, 1, 2, \ldots & (2.5)\\ &= \sum_n x[n-k]y[n]\ , k = 0, 1, 2, \ldots & (2.6) \end{aligned}$$

Note that since n was used as the time variable for x and y, we have introduced a new time variable, k, for r.

Suppose, for example, that we want to know the distance to a certain object, like an airplane. We transmit a radar pulse, $x[n]$, and receive a signal, $y[n]$, that contains the reflection of our pulse off of the object. For simplicity, let's assume that we know $y[n]$ is simply a delayed version of $x[n]$, that is[2],

$$y[n] = x[n - n_0], \tag{2.7}$$

However, we do not know the delay factor, n_0. Since n_0 is proportional to the distance to our object, this is the quantity that we wish to estimate. We can use correlation to help us determine this delay, but we need to use running correlation rather than simply in-place correlation.

Suppose that we first guess that n_0 is equal to zero. We correlate $y[n]$ with $x[n]$ and record the resulting correlation value as one sample of a new signal, $r[0]$. Then, we guess that n_0 is equal to one, shift $x[n]$ over by one sample, and correlate $y[n]$ with $x[n-1]$. We record this correlation value as $r[1]$. We can continue this shift-and-correlate procedure, building up the new signal $r[k]$ according to the formula

$$r[k] = C(x[n-k], y[n]) = \sum_{n=-\infty}^{\infty} x[n-k]y[n]. \tag{2.8}$$

[2]Recall that a signal $x[n-n_0]$ is equal to the signal $x[n]$ shifted n_0 samples to the right.

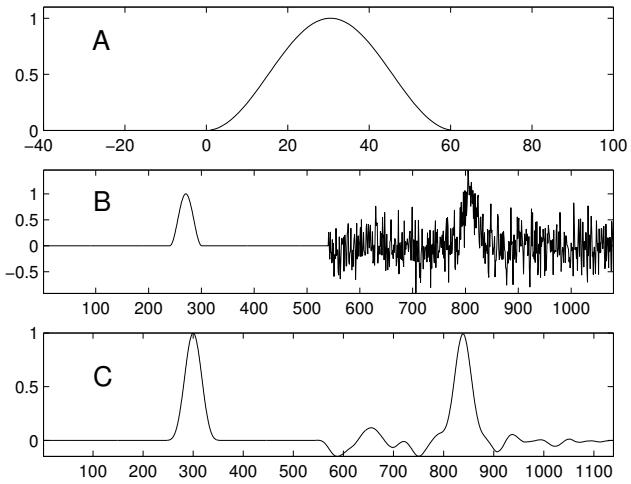


Figure 2.2: (A) A radar pulse. (B) A received sequence from the radar system, containing two pulses and noise. (C) The running correlation produced by correlating the radar pulse with the received signal.

Once we find a value of $r[k]$ that equals $E(x)$, we have found the value of n_0. This procedure of building up the signal $r[k]$ is known as *running correlation* or *sliding correlation*. We will refer to the resulting signal ($r[k]$ above) as the *correlation signal*.

As an example, Figure 2.2 shows a radar pulse, a received signal containing two delayed versions of the radar pulse (one without noise and one with noise), and the running correlation produced by correlating the pulse with the received signal.

Let us note a couple important features of the correlation signal. First, the limits of summation in equation (2.8) are infinite. Usually, though, the support of $x[n]$ and $y[n]$ will be finite, so we do not actually need to perform an infinite summation. Instead, the duration of the correlation signal will be equal to the sum of the durations of $x[n]$ and $y[n]$ minus one[3]. There will also be *transient effects* (or *edge effects*) at the beginning and end of the correlation signal. These transient effects result from cases where $x[n-k]$ only partially overlaps $y[n]$. Finally, notice that the value of the correlation signal at time $k = 0$ is just the in-place correlation $C(x[n], y[n])$.

2.2.3 Using correlation for signal detection

Whenever we wish to use correlation for signal detection, we use a two-part system. The first part of the system performs the correlation and produces the correlation value or correlation signal, depending upon whether we are doing in-place or running correlation. The second part of the system examines the correlation or correlation signal and makes a decision or sequence of decisions. See the block diagram given in Figure 2.3.

In the radar example used to motivate running correlation in Section 2.2.2, we simply checked to see if the correlation signal at a given point equals the energy of the transmitted signal. While this

[3]Suppose that support interval of $x[n]$ is $n_{x_1} \leq n \leq n_{x_2}$, while the support interval of $y[n]$ is $n_{y_1} \leq n \leq n_{y_2}$. For this general case, we can see that the first nonzero sample of $r[k]$ will occur at $k = n_{y_1} - n_{x_2}$. Similarly, the last nonzero sample will fall at $k = n_{y_2} - n_{x_1}$. Thus, the duration of $r[k]$ is $(n_{y_2} - n_{y_1}) + (n_{x_2} - n_{x_1}) + 1 = (n_{y_2} - n_{y_1} + 1) + (n_{x_2} - n_{x_1} + 1) - 1 = \text{duration}(x) + \text{duration}(y) - 1$.

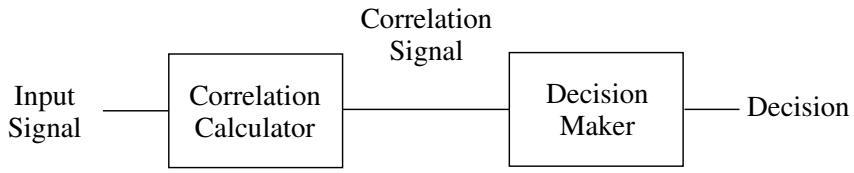


Figure 2.3: A generalized block diagram for a correlation-based detection system.

will work for the idealized system presented, real systems are usually much less ideal. We may have multiple reflections, distorted reflections, reductions in reflection amplitude, and various kinds of environmental noise. In order to address such problems in a wide variety of systems, we commonly use a simple threshold comparison as our decision maker. For instance, if we compute a running correlation signal $r[n]$, we might choose a constant c called a *threshold* and make a decision for each sample based on the following formula:

$$r \underset{0}{\overset{1}{\gtrless}} c \tag{2.9}$$

That is, when the correlation value r is greater than the threshold, c, we *decide* 1, or "signal present." If the value is less than the threshold, we decide 0, or "signal absent." In our radar example, for instance, we might select the threshold to be $c = E(x)/2$.

2.2.4 Using correlation for detection of signals transmitted simultaneously with other signals

Suppose that we wish to design a multi-user wireless communication system that permits several users to simultaneously transmit a sequence of message bits. That is, each user will transmit a signal across a common communication channel (for example, a wire or a small portion of the electromagnetic spectrum) that conveys his/her message bits, and despite the fact that these signals are received on top of one another, it should be possible to decode the message bits produced by any one of the users. The users of this system are completely uncoordinated; so no user has any idea who else might be using the system at any given time. How can we design a system so that each user can use the system without experiencing interference from the other users? This is the problem faced by the designers of cell phones and cordless phones, for example.

It turns out that we can use a correlation-based detector to address this problem. To begin, suppose that each user is trying to send a one bit message to a friend, and suppose each user has a distinct *code signal*, like those shown in Figure 2.4. Each code signal is made up of some number of binary *chips*, which are regions of constant signal value; the signals shown here each consist of ten chips.[4] To send a "one" message bit, the user transmits his or her code signal. To send a "zero" message bit, the user instead transmits a negated version of his or her code signal (which is is perfectly anticorrelated with the code signal).

Now, to send a *sequence* of message bits, the user concatenates these positive and negative versions of the code signal into a *sequence of code signals*, which is called the *transmitted signal* and which is input to the communication channel. Other users transmit their own message bits in

[4]Though the code signals clearly have a binary nature, we use the term "chip" to distinguish binary segments of the code signal from binary message elements, which we call "bits".

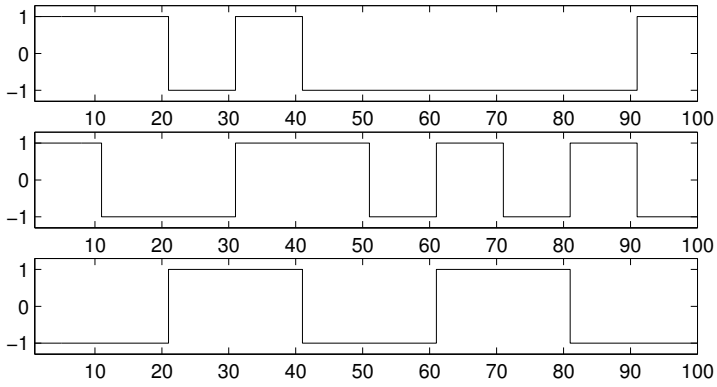


Figure 2.4: Example code signals for simultaneous communications

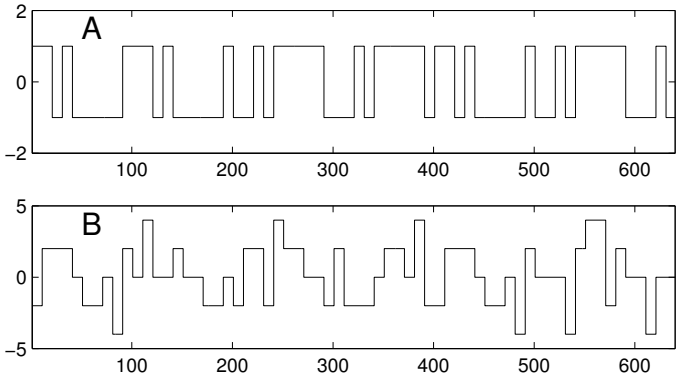


Figure 2.5: (A) Example of a transmitted signal. (B) The sum of the transmitted signals from four users.

the same fashion, except that, of course, they use different code signals. For example, Figure 2.5 shows a transmitted signal conveying eight message bits using the top code signal from Figure 2.4. It also shows this signal with the transmitted signals from three other users added to it. Notice that the signal in the upper panel is obscured in the lower panel.

When someone, say the *user's friend*, receives the signal from the communication channel[5] and wishes to *decode* the user's message bits, the friend correlates the received signal with the user's code signal. Specifically, in-place correlation of the received signal with the code signal produces a value with which a decision about the first message bit can be decided. Then in-place correlation of the received signal with a delayed version of the code signal produces a value from which the second message bit can be decided, and so on. Since each of these correlation values would equal plus or minus the energy of the code signal if there were no other signals or noise present, it is natural to have the decision maker decide that a message bit is "one" if the correlation value is positive and "zero" if the correlation value is negative. That is, a threshold of zero is chosen.

A communication system of this form is said to be a *code-division, multiple-access* (CDMA) system or a *direct-sequence, spread-spectrum* (DSSS) system. They are used, for example, in 900

[5]For simplicity, we assume the communication channel does not attenuate or otherwise distort the transmitted signal.

Mhz cordless telephones. Such systems work best when the code signals are as different as possible, i.e. when the normalized correlation between the code signals of distinct users are as near to zero as possible, which is what system designers typically attempt to do. Consider the examples in Figure 2.4. The first two code signals are completely uncorrelated, as are the second two. The first and third signals are slightly anticorrelated. The normalized correlation between these signals is only -0.2, which is small enough that these two code signals will not interfere much with one another.

Above, we've indicated that our detection system uses in-place correlation. This means that this system is *synchronous*; that is, the receiver knows when bits are sent. However, we can actually save ourselves some work by using running correlation, and then sampling the resulting correlation signal at the appropriate times. This is how we will implement this communication system in the laboratory assignment. Using the running correlation algorithm presented in this lab, the "appropriate times" occur in the correlation signal at the end of each code signal. That is, if our code signals are N samples long, we want to pick off the $(k \times N)^{th}$ sample out of the running correlator to decide the k^{th} message bit.

The threshold used to decode bits in this detection system, which we have chosen to be zero, is actually a design parameter of the system. If it should happen, for instance, that the system's noise is biased in a way that we tend to get slightly positive correlations when no signal is sent, then we would be able to improve performance of the system by using a positive threshold, rather than a threshold of zero. Alternatively, we might want to decide that no bit has been sent if the magnitude of the correlation is below some threshold. In this case, we actually have two thresholds. One separates "no signal" from a binary "one;" the other separates "no signal" from a binary "zero."

2.2.5 Noise, detector errors, and setting the threshold

Detectors, such as the radar and DSSS detectors we have discussed, must typically operate in the presence of noise. Here, we begin by discussing the radar example of Section 2.2.2, and conclude with a brief discussion of the DSSS detector.

When the radar pulse is $x[n]$, the typical received radar signal has the form

$$y[n] = x[n - n_0] + w[n] \tag{2.10}$$

where $x[n - n_0]$ is the reflected radar pulse and $w[n]$ is noise, i.e. an unpredictable, usually wildly fluctuating signal that normally is little correlated with $x[n]$ or any delayed version of $x[n]$. To estimate n_0, we perform running correlation of $y[n]$ with $x[n]$, and the resulting correlation signal is

$$\begin{aligned} r[k] &= \sum_{n=-\infty}^{\infty} x[n-k]y[n] \\ &= \sum_{n=-\infty}^{\infty} x[n-k](x[n-n_0] + w[n]) \\ &= \sum_{n=-\infty}^{\infty} x[n-k]x[n-n_0] + \sum_{n=-\infty}^{\infty} x[n-k]w[n] \\ &= r_0[k] + r_w[k] \ , \end{aligned} \tag{2.11}$$

where $r_0[k]$ is the running correlation of $x[n - n_0]$ with $x[n]$. Note that $r_0[k]$ is what $r[k]$ would be if there were no noise, as given in equation (2.8). $r_w[k]$ is the running correlation of the noise $w[n]$ with $x[n]$, which is added to $r_0[k]$. This shows that the effect of noise is to add $r_w[k]$ to $r_0[k]$.

Though in a well designed system $r_w[k]$ is usually close to zero, it will occasionally be large enough to influence the decision made by the decision maker.

In Section 2.2.3, we argued that a threshold-based decision maker was useful for such systems. Then, for example, when $r[k] > c$, the decision is that a radar pulse is present at time k, whereas when $r[k] < c$, the decision is that no radar pulse is present at time k. Since in the absence of noise $r[k] = E(x)$ when there is a radar pulse at time k, and since $r[k] = 0$ when there is no pulse at time k, it is natural, as mentioned in Section 2.2.3 to choose threshold $c = E(x)/2$.

Though it makes good sense to use a threshold detector, such a detector will nevertheless occasionally make an *error*, i.e. the wrong decision. Indeed, there are two types of errors that a detector can make. First, it could detect a reflection of the transmitted signal where no actual reflection exists. This is called a *false alarm*. It occurs at time k when $r_0[k] = 0$ and $r_w[k] > c$, i.e. when the part of the correlation due to noise is larger than the threshold. The other type of error occurs when the detector fails to detect an actual reflection because the noise causes the correlation to drop below the threshold even though a signal is present. This type of error is called a *miss*. It occurs when $r_0[k] = E(x)$ and $r[k] = E(x) + r_w[k] < c$, which in turn happens when $r_w[k] < c - E(x)$. In summary, a false alarm occurs when there is no radar pulse present, yet the noise causes $r_w[k] > c$, and a miss occurs when there is a radar pulse present, yet the noise causes $r_w[k] < c - E(x)$.

Depending on the detection system being developed, these two types of error could be equally undesirable or one could be more undesirable than another. For instance, in a defensive radar system, false alarms are probably preferable to misses, since the former are decidedly less dangerous. We can trade off the likelihood of these two types of error by adjusting the threshold. Raising the threshold decreases the likelihood of a false alarm, while lowering it decreases the likelihood of a miss.

It is often useful to know the frequency of each type of error. There is a simple way to empirically estimate these frequencies. First, we perform an experiment where we do not send any radar pulses, but simply record the received signal $y[n]$, which contains just environmental noise $w[k]$. We then compute its running correlation $r[k]$ with the radar pulse $x[n]$, which is just $r_w[k]$. We count the number of times that $r_w[k]$ exceeds the threshold c and divide by the total number of samples. This gives us an estimate the *false alarm rate*, which is the frequency with which the detector will decide a radar pulse is present when actually there is none. We can also use this technique to estimate the *miss rate*. When a radar pulse is present, an error occurs when $r_w[k] < c - E(x)$. Thus, we can estimate the *miss rate* by counting the number of times the already computed correlation signal $r_w[k]$ is less than $c - E(x)$, and dividing by the total number of samples.

The signal value distribution is also useful here. If we plot the histogram of values in $r_w[k]$, we can use this plot to determine the error rate estimates. The estimate of false alarm rate is the area of the histogram above values that exceeds c, divided by the total area of the histogram[6]. Similarly, the estimate of the miss rate is the area of the hisogram above values that are less than $c - E(x)$

Assuming that the distribution of $r_w[k]$ is symmetric about 0, we can minimize the *total error rate* (which is simply the sum of the false alarm and miss rates) by setting a threshold that yields the same number of false alarms as misses. Since the distribution of $r_w[k]$ is assumed to be symmetric, we get an equal number of false alarms and misses when $c = E(x)/2$, which is the threshold value suggested earlier.

Next, it is important to note how the error rates depend on the energy of the radar pulse. Consider first the false alarm rate, which corresponds to the frequency with which the noise induced correlation signal $r_w[k]$ exceeds $c = E(x)/2$. Suppose for example that the radar pulse is amplified by a factor of two. Then its energy increases by a factor of four, and consequently,

[6]That is, we sum the values in this region of the histogram and divide by the sum of all values in the histogram

the threshold c increases by a factor of four. On the other hand, one can see from the formula $r_w[k] = \int y[n]x[n-k]\,dx$ that the noise term $r_w[k]$ will be doubled. Since the threshold is quadrupled but the noise term is only doubled, the frequency with which the noise term exceeds the threshold will be greatly decreased, i.e. the false alarm rate is greatly decreased. A similar argument shows that the miss rate is also greatly decreased. Thus, we see that what matters is the energy of the radar pulse, in relation to the strength of the noise. If the energy of the signal increases, but the typical values of the noise $w[n]$ remain about the same, the system will make fewer errors. By making the energy sufficiently large, we can make the error rate as small as we like. In the lab assignments to follow, we will observe situations where the noise $w[n]$ is so strong that it completely obscures the radar pulse $x[n-n_0]$, yet the radar pulse is long enough that it has enough energy that a correlation detector will make few errors.

Finally, we comment on the effects of noise on the DSSS detector. In this case, instead of deciding whether a pulse is present or not, the detector decides whether a positive or negative code signal is present. As with the radar example, this must ordinarily be accomplished in the presence of noise. However, in this case there are two kinds of noise: environmental noise, similar to that which affects radar, and multiple user noise, which is due to other users transmitting their own code signals. In the absence of any noise, the in-place correlation $r(x,y)$ computed by the detector will be $+E(x)$ when the message bit is "one" and $-E(x)$ when the message bit is "zero" , where $x[n]$ is the user's code signal. For this reason, using a decision threshold $c = 0$ is natural. When the message bit is zero, an error occurs when $r(x,y) > 0$, which happens when the correlation term r_w due to noise exceeds $E(x)$. Similarly, when the message bit is one, an error occurs when $r(x,y) < 0$, which happens when $r_w < -E(x)$. As with the radar example, errors occur less frequently when the signal energy becomes larger. This will be evident in the lab assignment, when code signals of different lengths, and hence different energies, are used.

2.2.6 An algorithm for running correlation

Here, we provide an algorithm for running correlation. One of its primary benefits is that it is easy to understand. In this algorithm, we imagine the filter as a box into which we drop one new sample of the "incoming" signal and a corresponding new sample of the correlation signal comes out. This allows the algorithm to be used in real-time: as samples of our signal arrive (from a radar detector, for instance), we can process the resulting signal with almost no delay.

In this algorithm we refer to the signal we are looking for (i.e., the transmitted radar signal) as $x[n]$, following (2.8). The algorithm goes like this:

1. Initialize an *input buffer*, which is simply an array with length equal to the duration of $x[n]$, to all zeros.
2. For each sample that comes in:
   (a) Update the buffer by doing the following:
      i. Discard the sample at the beginning of the buffer.
      ii. Shift the rest of the samples one place towards the beginning of the buffer.
      iii. Insert the incoming sample at the end of the buffer.
   (b) Initialize a running sum variable to zero.
   (c) For each position, n, in the buffer:
      i. Multiply the n^{th} position in the input buffer by the n^{th} sample of $x[n]$.
      ii. Add the resulting product to the running sum.

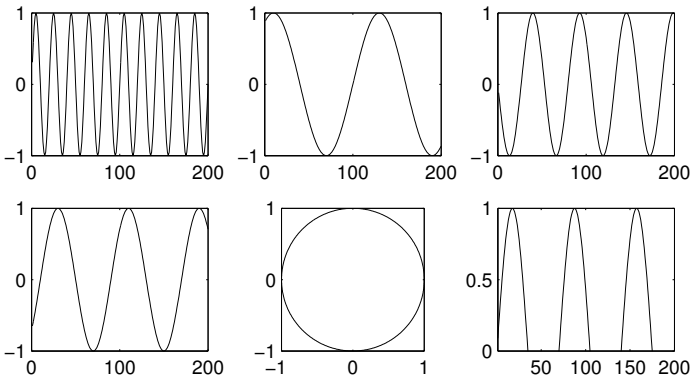


Figure 2.6: Example of a MATLAB figure with subplots.

(d) Output the running sum as the next sample of the correlation signal.

In the laboratory assignment, you will be asked to complete an implementation of this algorithm. Note that significant portions of this algorithm can be implemented very simply in MATLAB. For instance, all of (a) can be accomplished using a single line of code. Similarly, parts (b) through (d) can all be accomplished in a single line using one of MATLAB's built-in functions and its vector arithmetic capabilities.

2.3 Some MATLAB commands for this lab

- **Calculating in-place correlation:** If you have two signals, `x` and `y`, that you wish to correlate, simply use the command

  ```
  >> c_xy = sum(x.*y);
  ```

  Note that `x` and `y` must be the same size; otherwise MATLAB will return an error.

- **The `subplot` command:** In order to put several plots on the same figure in MATLAB, we use the `subplot` command. `subplot` creates a rectangular array of axes in a figure. Figure 2.6 has an example figure with such an array. Each time you call `subplot`, you activate one of the axes. `subplot` takes three input parameters. The first and second indicate the number of axes per row and the number of axes per column, respectively. The third parameter indicates which of the axes to activate by counting along the rows[7]. Thus the command:

  ```
  >> subplot(2,3,5)
  ```

  activates the plot with the circle in Figure 2.6.

- **The `axis` command:** The command `axis` allows us to set the axis range for a particular plot. Its single input argument is a vector of the form `[ x_min, x_max, y_min, y_max ]`. For instance, if you wish to change the display range of the currently active plot (or subplot) so that the x-axis ranges from 5 and 10 and the y-axis ranges from -100 to 100, simply execute the command

[7] Note in particular that this is the opposite of MATLAB's usual convention!


```
>> axis([ 5, 10, -100, 100]);
```

Other useful forms of the `axis` command include `axis tight`, which fits the axis range closely around the data in a plot, and `axis equal`, which assures that the x- and y-axes have the same scale.

- **Buffer operations in MATLAB:** It is often useful to use MATLAB's vectors as *buffers*, with which we can shift values in the buffer towards the beginning or end of the buffer by one position. Such an operation has two parts. First, we discard the number at the beginning or end of the buffer. If our buffer is a vector `b`, we can do this using either `b = b(2:end)` or `b = b(1:end-1)`. Then, we append a new number to the opposite end of the buffer using a standard array concatenation operation. Note that we can easily combine these two steps into a single command. For instance, if `b` is a row vector and we wish to shift towards the end of the buffer, we use the command

  ```
  >> b = [ new_sample, b(1:end-1) ];
  ```

- **Counting elements that meet some condition:** Occasionally we may want to determine how many elements in a vector meet some condition. This is simple in MATLAB because of how the conditional operators are handled. Recall that for a vector, `v`, `(v == 3)` will return a vector with the same size as `v`, the elements of which are either 1 or 0 depending upon the truth of the conditional statement. Thus, to count the number of elements in `v` that equal 3, we can simply use the command

  ```
  >> count = sum(v == 3);
  ```

2.4 Demonstrations in the Lab Section

- Detector error types
- Why use correlation? Or, when energy detectors break down.
- "In-place" correlation as a similarity measure
- Running correlation
- Multi-user communication

2.5 Laboratory Assignment

In this lab assignment, all signals are discrete-time and their support is assumed to be of the form $\{1, 2, \ldots, N\}$.

1. (Computing and interpreting in-place correlations) Download the file `code_signal.m` and use it to create the following signals:

   ```
   >> code1 = code_signal(75,10);
   >> code2 = code_signal(50,10);
   >> code3 = code_signal(204,10);
   ```


(a) (Plotting code signals) Use `subplot` and `stairs` to plot the three code signals on three separate axes in the same figure. After plotting each signal, call `axis([1, 100, -1.5, 1.5])` to make sure that the signal is visible.

- Include your figure, with axis labels on *each subplot*, a figure number and caption, and the generating code in your report.

(b) (Calculate statistics) For each of the three signals generated above, calculate:

- Their mean values.
- Their energies.

(c) (Calculate correlations) Calculate the "in-place" correlation and normalized correlation for the following pairs of signals.

- `code1` and `code2`
- `code1` and `code3`
- `code2` and `code3`

(d) (Classify correlations) For each of the signal pairs given in problem 1c:

- Identify each pair as positively correlated, uncorrelated, or negatively correlated.

2. (Implementing and interpreting running correlation) Download the file `run_corr.m`, which is a "skeleton" file for an implementation of the "real-time" running correlation algorithm described in Section 2.2.2. It accepts two input signals, performs running correlation on them, and produces the correlation signal with a length equal to the sum of the lengths of the input signals minus one.

(a) (Write the code) Complete the function, following the algorithm given in Section 2.2.2. You can use the completed demo version of the function, `run_corr_demo.dll` to check your function's output[8].

- Include your code in the MATLAB appendix of your report.

(b) (Compute running correlations) Use `run_corr.m` to compute the running correlation between the following pairs of signals, and plot the resulting correlation signals on the same figure using `subplot`.

- `code1` and `code2`.
- `code3` and itself.

(c) (Interpret a running correlation) When performing running correlation with a signal and itself, the resulting correlation signal has some special properties. Look at the correlation signal that you computed between `code3` and itself.

- Is the correlation signal symmetric? (It can be shown that it should be.)
- What is the maximum value of the correlation signal? How does the maximum value relate relate to the energy of `code3`?

3. (Using correlation to decode DSSS signals transmitted simultaneously with other signals.) Download the file `lab2_data.mat` and load it into your workspace. The file contains the variable which represents a received signal that is the sum of several message carrying signals, one from each of four users. The message carrying signal from each user conveys a sequence

[8] If you cannot get your function working properly, you may use `run_corr_demo.dll` to complete the rest of the assignment.

of bits using the direct-sequence spread-spectrum technique, described in Section 2.2.4, in which each bit is conveyed by sending a code signal or its negative. Each user has a different code signal. One of the code signals is the ten chip signal corresponding to the integer 170, while another is the six chip signal corresponding to the integer 25. The other two code signals are unknown to us. In this problem, we will try to extract the bit sequences conveyed by the known code signals from `dsss`. Start by generating the following code signals:

```
>> cs1 = code_signal(170,10);
>> cs2 = code_signal(25,6);
```

(a) (Plot the signals) First, let's look at the signals we're given.

- Use `subplot` and `stairs` to plot `dsss`, `cs1`, and `cs2` on three separate axes of the same figure.

(b) (Decoding the bits of the user with the longer code signal) Start by using `run_corr` to correlate the received signal `dsss` with the longer code signal `cs1`. Call the resulting signal `cor1`. Now, to decode the sequence of message bits from this user, we need to extract the appropriate samples from `cor1`. That is, we need to extract just those samples of the running correlation that correspond to the appropriate in-place correlations. We can do this in MATLAB using the following command:

```
>> sub_cor1 = cor1(length(cs1):length(cs1):length(cor1));
```

Each sample of `sub_cor1` is used to make the decision about one of the user's bits. When it is greater than zero, i.e. the correlation of the received signal with the code signal is positive, the decoder decides the bit is 1. When it is less than zero, the decoder decides the user's bit is 0.

- On two subplots of the same figure, use `plot` to plot `cor1`, and `stem` to plot `sub_cor1`.
- Decode the sequence of bits. (You can do this visually or with MATLAB ☺) (Hint: The sequence is 10 bits long, and the first 3 bits are "011".)

(c) (Decoding the bits of the user with the shorter code signal) Repeat the procedure in a and b above, this time using the code signal `cs2`. Call your correlation signal `cor2`, and the vector of extracted values `sub_cor2`.

- On two subplots of the same figure, use `plot` to plot `cor2`, and `stem` to plot the signal `sub_cor2`.
- Decode the sequence of bits. (Hint: there are 17 bits in this sequence.)
- Since the code signal `cs2` has less energy (because it is shorter), there is a greater chance of error. Are there any decoded bits that you suspect might be incorrect? Which ones? Why?

4. (Using running correlation to detect reflected radar pulses) `lab2_data.mat` also contains three other signals: `radar_pulse`, `radar_received`, and `radar_noise`. The received signal contains several reflections of the transmitted radar pulse and noise. The signal `radar_noise` contains noise with similar characteristics to the noise in the received signal without the reflected pulses.

(a) (Examining the radar signals) First, let's take a look at the first two signals.

- Calculate the energy of `radar_pulse`, $E(x)$.
- Use `subplot` to plot `radar_pulse` and `radar_received` in separate axes of the same figure.
- Can you identify the reflected pulses in the received signal by visual inspection alone?

(b) (Perform running correlation) Use `run_corr` to correlate `radar_received` with `radar_pulse`.

- Plot the resulting correlation signal.
- Where are the received pulses? Visually identify sample locations of each pulse in the correlation signal.
- Compare the heights of the peaks to the energy of the radar pulse and explain why the peaks are larger/smaller than the energy.
- Given that the speed of light is 3×10^8 m/s and the sampling frequency of the detector is 10^7 samples per second, what is the approximate distance to each object[9]?

(c) (Estimating error rates, as in Section 2.2.5) In a real radar detector, the correlation signal would be compared to a threshold to perform the detection. In order to estimate the error rates for such a detector, let's consider a threshold that is equal to one-half the energy of the transmitted pulse, i.e. $c = E(x)/2$. Perform running correlation between `radar_pulse` and `radar_noise` call the resulting correlation signal `noise_c`.

- Plot `noise_c`.
- For how many samples is `noise_c` *greater* than this threshold? Use this value to estimate the false alarm rate.
- For how many samples is `noise_c` *less* than this threshold minus the energy of the transmitted pulse? Use this value to estimate the miss rate.
- What is the total error rate for this threshold?

(d) (Determining error rates from a histogram) As discussed in Section 2.2.5, we can use a histogram to judge the number of errors as well.

- Plot the histogram of `noise_c` using 100 bins.
- Describe how you could derive the error numbers in problem 4c from the histogram.

(e) (Setting the threshold to achieve a particular error rate) Suppose that detector false alarms are considered to be more serious than detector misses. Thus, we have determined that we want to raise the threshold so that we achieve a false alarm rate of approximately 0.004. Find a threshold that satisfies this requirement.

- What is your threshold?
- What is the false alarm rate on this noise signal with your threshold?
- What is the miss rate on this noise signal with your threshold?
- What is the total error rate for the new threshold? Compare this to the total error rate of the threshold used in problem 4c.

5. On the front page of your report, please provide an estimate of the average amount of time spent outside of lab by each member of the group.

[9] Remember that the radar pulse must travel to the object and then back again.

Laboratory 3

Sinusoids and Sinusoidal Correlation

3.1 Introduction

Sinusoids are important signals. Part of their importance comes from their prevalence in the everyday world, where many signals can be easily described as a sinusoid or a sum of sinusoids. Another part of their importance comes from their properties when passed through linear time-invariant systems. Any linear time-invariant system whose input is a sinusoid will have an output that is a sinusoid of the same frequency, but possibly with different amplitude and phase. Since a great many natural systems are linear and time-invariant, this means that sinusoids form a powerful tool for analyzing systems.

Being able to identify the parameters of a sinusoid is a very important skill. From a plot of the sinusoid, any student of signals and systems should be able to easily identify the amplitude, phase, and frequency of that sinusoid.

However, there are many practical situations where it is necessary to build a system that identifies the amplitude, phase, and/or frequency of a sinusoid — not from a plot, but from the actual signal itself. For example, many communication systems convey information by *modulating*, i.e. perturbing, a sinusoidal signal called a *carrier*. To *demodulate* the signal received at the antenna, i.e. to recover the information conveyed in the transmitted signal, the receiver often needs to know the amplitude, phase, and frequency of the carrier. While the frequency of the sinusoidal carrier is often specified in advance, the phase is usually not specified (it is just whatever phase happens to occur when the transmitter is turned on), and the amplitude is not known because it depends on the attenuation that takes place during transmission, which is usually not known in advance. Moreover, though the carrier frequency is specified in advance, no transmitter can produce this frequency exactly. Thus, in practice the receiver must be able to "lock onto" the actual frequency that it receives.

Doppler radar provides another example. With such a system, a transmitter transmits a sinusoidal waveform at some frequency f_o. When this sinusoid reflects off a moving object, the frequency of the returned sinusoid is shifted in proportion to the velocity of the object. A system that determines the frequency of the reflected sinusoid will also be able to determine the speed of the moving object.

How can a system be designed that automatically determines the amplitude, frequency and phase of a sinusoid? One could imagine any number of heuristic methods for doing so, each based on how you would visually extract these parameters. It turns out, though, that there are more convenient methods for doing so – methods which involve correlation. In this lab, we will examine how to automatically extract parameters from a sinusoid using correlation. Along the way, we will discover how complex numbers can help us with this task. In particular, we will make use of the *complex*

exponential signal and see the mathematical benefits of using an "imaginary" signal that does not really exist.

3.1.1 "The Question"

- How can we design a system that automatically determines the amplitude and phase of a sinusoid with a known frequency?
- How can we design a system that automatically determines the frequency of a sinusoid?

3.2 Background

3.2.1 Complex numbers

Before we begin, let us quickly review the basics of complex numbers. Recall the a complex number $z = x + jy$ is defined by its *real part*, x, and its *imaginary part*, y, where $j = \sqrt{-1}$. Also recall that we can rewrite any complex number into *polar form*[1] or *exponential form*, $z = re^{j\theta}$, where $r = |z|$ is the *magnitude* of the complex number and $\theta = \text{angle}(z)$ is the *angle*. We can convert between the two forms using the formulas

$$x = r\cos(\theta) \tag{3.1}$$
$$y = r\sin(\theta) \tag{3.2}$$

and

$$r = \sqrt{x^2 + y^2} \tag{3.3}$$
$$\theta = \begin{cases} \tan^{-1}\left(\frac{y}{x}\right), & x \geq 0 \\ \tan^{-1}\left(\frac{y}{x}\right) + \pi, & x < 0 \end{cases} \tag{3.4}$$

A common operation on complex numbers is the *complex conjugate*. The complex conjugate of a complex number, z^*, is given by

$$z^* = x - jy \tag{3.5}$$
$$= re^{-j\theta} \tag{3.6}$$

Conjugation is particularly useful because $zz^* = |z|^2$.

Euler's[2] formula is a very important (and useful) relationship for complex numbers. This formula allows us to relate the polar and rectangular forms of a complex number. Euler's formula is

$$e^{j\theta} = \cos(\theta) + j\sin(\theta) \tag{3.7}$$

Equally important are Euler's inverse formulas:

$$\cos(\theta) = \frac{e^{j\theta} + e^{-j\theta}}{2} \tag{3.8}$$
$$\sin(\theta) = \frac{e^{j\theta} - e^{-j\theta}}{2j} \tag{3.9}$$

It is *strongly recommended* that you commit these three equations to memory; you will be using them regularly throughout this course.

[1] Sometimes the polar form is written as $z = r\angle\theta$, which is a mathematically less useful form. This form, however, is useful for suggesting the interpretation of r as a radius and θ as an angle.

[2] Pronounced "oiler's".

3.2.2 Sinusoidal and complex exponential signals in continuous time

Recall that a continuous-time sinusoid in *standard form*, $s(t)$, is given by the formula

$$s(t) = A\cos(\omega_0 t + \phi), \tag{3.10}$$

where $A > 0$ is the sinusoid's *amplitude*, ω_0 is the sinusoid's *frequency* given in *radian frequency* (radians per second), and ϕ is the sinusoid's *phase*. It is also common to represent such a sinusoid in the following form

$$s(t) = A\cos(2\pi f_0 t + \phi), \tag{3.11}$$

where f_0 is the sinusoid's frequency given in Hertz (Hz, or cycles per second). Note that $\omega_0 = 2\pi f_0$. The frequency of a sinusoid is generally restricted to be positive.

The notation for sinusoids also extends to a special signal known as the *complex exponential signal*[3]. Complex exponential signals are very similar to sinusoids, and have the same three parameters. We define a continuous-time complex exponential signal, $c(t)$, in standard form as

$$c(t) = Ae^{j(\omega_0 t+\phi)} \tag{3.12}$$

It is generally useful to consider that sinusoids are composed of a sum of complex exponential signals by using Euler's inverse formulas. Thus, a sinusoid in standard form can be rewritten in several different ways:

$$\begin{aligned} s(t) &= A\cos(\omega_0 t + \phi) && (3.13)\\ &= \frac{A}{2}\left[e^{j(\omega_0 t+\phi)} + e^{-j(\omega_0 t+\phi)}\right] && (3.14)\\ &= \frac{A}{2}(c(t) + c^*(t)) && (3.15)\\ &= \text{Re}\left\{Ae^{j(\omega_0 t+\phi)}\right\} && (3.16) \end{aligned}$$

Using Euler's formula, we can also interpret a complex exponential signal $c(t)$ as the sum of a real cosine wave and an imaginary sine wave:

$$c(t) = A\cos(\omega_0 t + \phi) + jA\sin(\omega_0 t + \phi) \tag{3.17}$$

Sometimes it is useful to visualize a complex exponential signal as a "corkscrew" in three dimensions, as in Figure 3.1. Note that it is common to permit complex exponential signals to have either positive or negative frequency. The sign of the frequency determines the "handedness" of the corkscrew.

3.2.3 Finding the amplitude and phase of a sinusoid with known frequency

We've suggested that we can use *correlation* to help us determine the amplitude and phase of a sinusoid with known frequency. Suppose that we have a continuous-time sinusoid (the *target sinusoid*)

$$s(t) = A\cos(\omega_0 t + \phi) \tag{3.18}$$

with known frequency ω_0, but unknown amplitude A and phase ϕ, which we would like to find. We can perform in-place correlation[4] between this sinusoid and a *reference sinusoid*, $u(t)$, with the

[3] These are sometimes referred to simply as *complex exponentials*.

[4] In-place correlation between two real, continuous-time signals, $x(t)$ and $y(t)$ is defined as $C(x, y) = \int_a^b x(t)y(t)dt$. The length $(b - a)$ is the *correlation length*.

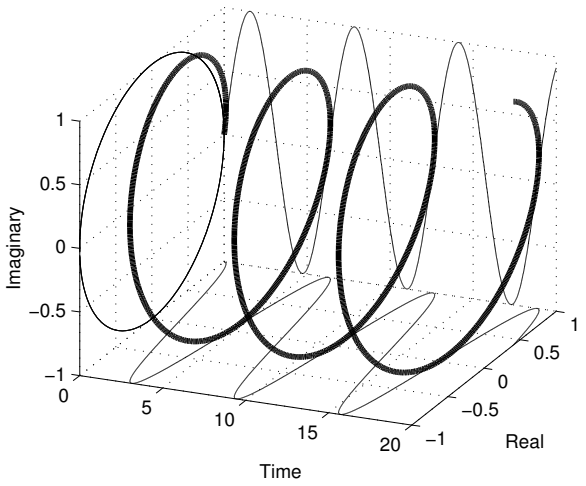


Figure 3.1: Three-dimensional plot of a complex exponential signal.

same frequency and known amplitude and phase. Without loss of generality, let $u(t)$ have $A = 1$ and $\phi = 0$. Then[5],

$$C(s,u) = \int_{t_1}^{t_2} A\cos(\omega_0 t + \phi)\cos(\omega_0 t)dt \tag{3.19}$$

$$= \frac{A}{2}\int_{t_1}^{t_2} \cos(\phi) + \cos(2\omega_0 t + \phi)dt \tag{3.20}$$

$$= \frac{A}{2}\left[\cos(\phi)t + \frac{1}{4\omega_0}\sin(\omega_0 t + \phi)\right]_{t_1}^{t_2} \tag{3.21}$$

Since we know the frequency, ω_0, we can easily set the limits of integration to include an integer number of fundamental periods of our sinusoids. In this case, the second term evaluates to zero and the correlation reduces to

$$C(s,u) = \frac{A}{2}\cos(\phi)(t_2 - t_1) \tag{3.22}$$

This formula is a useful first step. If we happen to know the phase ϕ, then we can readily calculate the amplitude A of $s(t)$ from $C(s, u0$. Similarly, if we know the amplitude A, we can narrow the phase ϕ down to one of two values. If both amplitude and phase are unknown, though, we cannot uniquely determine them.

Note that if the interval over which we correlate is not a multiple of the fundamental period of $u(t)$, then the second term in equation (3.21) will not be zero. However, if as commonly happens ω_0 is much greater than one, then the second term will be so small that it can be ignored, and equation (3.22) holds with approximate equality.

To resolve the ambiguity when both amplitude and phase are unknown, one common approach to correlate with a second reference sinusoid that is $\frac{\pi}{2}$ out of phase with the first. Here, though, we will explore a different method which is somewhat more enlightening. Notice what happens if we

[5]Recall that $\cos(A)\cos(B) = \frac{1}{2}\cos(A-B) + \frac{1}{2}\cos(A+B)$.

use a complex exponential,

$$c(t) = e^{j\omega_0 t} \tag{3.23}$$

as our reference signal[6]:

$$\begin{aligned} C(s,c) &= \int_{t_1}^{t_2} s(t)c^*(t)\,dt && (3.24)\\ &= \int_{t_1}^{t_2} A\cos(\omega_0 t+\phi)e^{-j\omega_0 t}\,dt && (3.25)\\ &= \int_{t_1}^{t_2} \frac{A}{2}\left[e^{j(\omega_0 t+\phi)}+e^{-j(\omega_0 t+\phi)}\right]e^{-j\omega_0 t}\,dt && (3.26)\\ &= \frac{A}{2}\int_{t_1}^{t_2} e^{j\phi}+e^{-j(2\omega_0 t+\phi)}\,dt && (3.27)\\ &= \frac{A}{2}\left[e^{j\phi}t+\frac{-1}{2j\omega_0}e^{-j(2\omega_0 t+\phi)}\right]_{t_1}^{t_2} && (3.28) \end{aligned}$$

If we again assume that we are correlating over an integer number of periods of our target sinusoid, then the second term goes to zero and we are left with

$$C(s,c) = \frac{A}{2}\,e^{j\phi}(t_2-t_1). \tag{3.29}$$

Our correlation has resulted in a simple complex number whose magnitude is directly proportional to the amplitude of the original sinusoid and whose angle is identically equal to its phase! We can easily turn the above formula inside-out to obtain

$$\begin{aligned} A &= \frac{2}{t_2-t_1}\,|C(s,c)| && (3.30)\\ \phi &= \text{angle}(C(s,c)) && (3.31) \end{aligned}$$

We can also see from equation (3.29) that in correlating with a complex exponential signal, we have effectively calculated the phasor[7] representation of our sinusoid.

As with the case of correlating with a sinusoid, we note that when the interval over which we correlate is not a multiple of the fundamental period of $c(t)$, then the second term in equation (3.28) is not zero. However, if as commonly happens ω_0 is much greater than 1, then the second term will again be small enough that it can be ignored, and equations (3.29), (3.30), and (3.31) hold with approximate equality.

The Amplitude and Phase Calculator

In this lab we will implement a system that estimates the amplitude and phase of a sinusoid with a known frequency. Since we will do this using a computer and MATLAB, we must necessarily work with sampled version of the signals $s(t)$ and $c(t)$. Specifically, if T_s denotes the sampling period, then we work with the discrete-time signals

$$\begin{aligned} s[n] &= s(nT_s) = A\cos(\omega_0 T_s n+\phi) && (3.32)\\ c[n] &= c(nT_s) = e^{j\omega_0 T_s n} && (3.33) \end{aligned}$$

[6]Notice that we conjugate our complex exponential here. This is because correlation between two *complex* signals is defined as $\int x(t)y^*(t)dt$.

[7]When we represent a sinusoid with amplitude A and phase ϕ as the complex number $Ae^{j\phi}$ to simplify the calculation of a sum of two or more sinusoids, this complex number is known as a *phasor*.

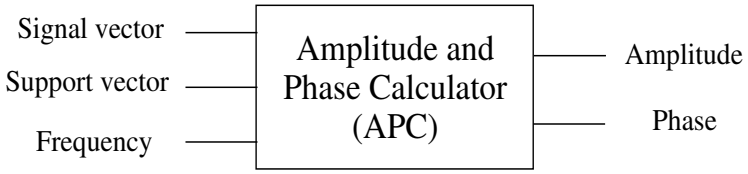


Figure 3.2: System diagram for the "amplitude and phase calculator."

As shown below, when T_s is small, the correlation between $s(t)$ and $c(t)$ can be approximately computed from the correlation between $s[n]$ and $c[n]$. Let $\{n_1, \ldots, n_2\}$ denote the discrete-time interval corresponding to the continuous-time interval $[t_1, t_2]$, and let $N = n_2 - n_1 + 1$ denote the number of samples taken in the interval $[t_1, t_2]$, so that $t_2 - t_1 \approx NT_s$. Then,

$$\begin{aligned} C(s,c) &= \int_{t_1}^{t_2} s(t)\, c^*(t)\, dt && (3.34)\\ &= \sum_{n=n_1}^{n_2} \int_{nT_s}^{(n+1)T_s} s(t)\, c^*(t)\, dt && (3.35)\\ &\approx \sum_{n=n_1}^{n_2} \int_{nT_s}^{(n+1)T_s} s(nT_s)\, c^*(nT_s)\, dt && (3.36)\\ &= \sum_{n=n_1}^{n_2} s(nT_s)\, c^*(nT_s)\, T_s && (3.37)\\ &= \sum_{n=n_1}^{n_2} s[n]\, c^*[n]\, T_s && (3.38)\\ &= C_d(s,c)\, T_s && (3.39) \end{aligned}$$

where the approximation leading to the third relation is valid because T_s is small, and consequently the signals $s(t)$ and $c(t)$ change little over each T_s second sampling interval, and where we use $C_d(s,c)$ to denote the correlation between the discrete-time signals $s[n]$ and $c[n]$, to distinguish it from the correlation between continuous time signals $s(t)$ and $c(t)$. We see from this derivation that the continuous-time correlation is approximately the discrete-time correlation multiplied by the sampling interval, i.e.

$$C(s,c) \approx C_d(s,c)T_s \qquad (3.40)$$

We will use this value of correlation in equations (3.30) and (3.31) to estimate the amplitude and phase of a continuous-time sinusoid.

In the laboratory assignment, we will be implementing an "amplitude and phase calculator" (APC) as a MATLAB function. A diagram of this system is shown in Figure 3.2. The system takes three input parameters. The first is the *signal vector* which contains the sinusoid itself. The second is the *support vector* for the sinusoid. The third input parameter is the frequency of the reference sinusoid in radians per second. Note that for the system's output to be exact, the input sinusoid must be defined over exactly an integer number of fundamental periods.

The system outputs the sinusoid's amplitude and its phase in radians. The system calculates these outputs by first computing the in-place correlation given by equations (3.37) or (3.38). Then,

this correlation value is used with equations (3.30) and (3.31) to compute the amplitude and phase. Note that in equation (3.30), we need to replace $t_2 - t_2$ with $N = n_2 - n_1 + 1$ when implementing in discrete time.

3.2.4 Determining the frequency of a target sinusoid

Suppose now that we are given the task of making a system that automatically determines the frequency of a target sinusoid. It turns out that correlation can help us with this problem as well. Consider the following case. Let the target sinusoid be defined by $s(t) = A\cos(\omega_s t + \phi)$, where ω_s, A, and ϕ are all unknown. We correlate $s(t)$ with a complex exponential signal, $c(t) = e^{j\omega_c t}$, with frequency ω_c, where $\omega_s \neq \omega_c$:

$$\begin{aligned} C(s,c) &= \int_{t_1}^{t_2} s(t)c^*(t)\,dt && (3.41)\\ &= \int_{t_1}^{t_2} A\cos(\omega_s t+\phi)e^{-j(\omega_c t)}dt && (3.42)\\ &= \int_{t_1}^{t_2} \frac{A}{2}\left[e^{j(\omega_s t+\phi)} + e^{-j(\omega_s t+\phi)}\right] e^{-j(\omega_c t)}\,dt && (3.43)\\ &= \frac{A}{2}\int_{t_1}^{t_2} e^{j[(\omega_s-\omega_c)t+\phi]} + e^{-j[(\omega_s+\omega_c)t+\phi]}\,dt && (3.44)\\ &= \frac{A}{2}\left[\frac{1}{\omega_s-\omega_c}e^{j[(\omega_s-\omega_c)t+\phi]} + \frac{1}{\omega_s+\omega_c}e^{-j[(\omega_s+\omega_c)t+\phi]}\right]_{t_1}^{t_2} && (3.45)\end{aligned}$$

Here, let us make a simplifying assumption and assume that $(\omega_s + \omega_c)$ is sufficiently large that we can neglect the second term. Then, we have

$$C(s,c) \approx \frac{A}{2(\omega_s-\omega_c)}\left[e^{j[(\omega_s-\omega_c)t_2+\phi]} - e^{-j[(\omega_s-\omega_c)t_1+\phi]}\right] \quad (3.46)$$

The resulting equation depends primarily on the frequency difference $(\omega_s - \omega_c)$ between the target sinusoid and our reference signal. Though it is not immediately apparent, the value of this correlation converges to the value of equation (3.29) as the $(\omega_s - \omega_c)$ approaches zero.

Consider now the length-normalized correlation, $\widetilde{C}(s,c)$, defined as

$$\widetilde{C}(s,c) = \frac{C(s,c)}{t_2 - t_1}. \quad (3.47)$$

One can see from equation (3.29) that when the reference and target signals have the same frequency, the length-normalized correlation does not depend on the length of the signal. However, when the signals have different frequencies, one can see from equations (3.46) and (3.47) that the magnitude of the length-normalized correlation becomes smaller as we correlate over a longer period of time. (This happens more slowly as the frequency difference becomes smaller.) In the limit as the correlation length goes to infinity, *the length-normalized correlation goes to zero unless the frequencies match exactly*. This is a very important theoretical result in signals and systems.

Another special case occurs when we correlate over a *common period* of the target and reference signals. This occurs when our correlation interval includes an integer number of periods of *both* the target signal and reference signal. In this case, the correlation in equation (3.46), for signals of

different frequencies, is identically zero[8]. Of course, the correlation is *not* zero when the frequencies match. Note that this is the same condition required for equation (3.29) to be exact.

How does all of this help us to determine the frequency of the target sinusoid? The answer is perhaps less elegant than one might hope; basically, we "guess and check". If we have no prior knowledge about possible frequencies for the sinusoid, we need to check the correlation with complex exponentials having a variety of frequencies. Then, whichever complex exponential yields the highest correlation, we take the frequency of that complex exponential as our estimate of the frequency of the target signal. In the next section, we will formalize this algorithm.

A frequency estimation algorithm

Suppose that we have a continuous-time target sinusoid $s(t)$ with support $[0, T]$ with unknown amplitude, frequency, and phase. To estimate these parameters, we'll calculate the length-normalized correlation between this signal and reference complex exponentials with various frequencies over the signal's T second length. Then, we look for the frequency that produces the largest correlation.

We choose the frequencies of these complex exponentials to be multiples $1/T$ so that the correlation is over an integer number of periods of each complex exponential. That is, the frequencies will be $1/T, 2/T, \ldots$. As in the previous subsection, we'll need to approximately compute the correlation from samples of $s(t)$ and each reference exponential, taken with some small sampling interval T_s. For convenience we'll take N samples and choose $T_s = T/N$ for some large even integer N. With samples taken at intervals of T_s seconds, we cannot hope to do a good job of correlating with complex exponentials with very high frequency. The rule of thumb that you will learn in Chapter 4 is that, at the very least, two samples are needed from each period of the signal being sampled. Therefore, the highest frequency with which we will correlate is, approximately, $1/(2T_s)$. Specifically, we will correlate $s(t)$ with complex exponentials at frequencies

$$\frac{1}{T},\ \frac{2}{T},\ \cdots,\ \frac{N}{2T} = \frac{1}{2T_s} \tag{3.48}$$

Then, for $k = 1, 2, \ldots, N/2$, the length normalized correlation of $s(t)$ with the complex exponential at frequency $\frac{k}{T}$ is (using equations (3.37), (3.38) and (3.47))

$$X[k] \approx \frac{1}{T}\sum_{n=0}^{N-1} s(nT_s)\, e^{-j2\pi\frac{k}{T}nT_s} T_s \tag{3.49}$$

$$= \frac{1}{N}\sum_{n=0}^{N-1} s[n]\, e^{-j2\pi\frac{k}{N}n} \tag{3.50}$$

where we have used the fact that $T_s/T = 1/N$ and where we have denoted the result $X[k]$ because this is the notation used in future labs for the last formula above. Thus, the output output of these correlations is the set of $N/2$ numbers $X[1], \ldots, X[N/2]$. Remember that $X[k]$ will generally be complex. To estimate the frequency of the target sinusoid, we simply identify the value of k for which $|X[k]|$ is largest. With k_{max} denoting this value, our estimated frequency, ω_{est}, is

$$\omega_{est} = 2\pi\frac{k_{max}}{T} = 2\pi\frac{k_{max}}{NT_s} \tag{3.51}$$

Now that we have estimated the frequency, we should also be able to estimate the amplitude and phase as well. In fact, we have almost calculated these estimates already. From equations (3.30) and

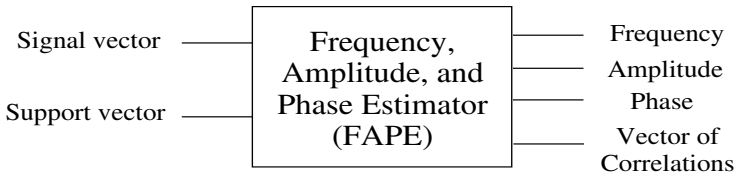


Figure 3.3: System diagram for the "frequency, amplitude, and phase estimator."

(3.31), they are:

$$A_{est} = 2|X[k_{max}]| \tag{3.52}$$
$$\phi_{est} = \text{angle}(X[k_{max}]) \tag{3.53}$$

There is one potential problem here, however. Previously, we assumed the frequency was known exactly when determining the amplitude and phase; now, we only know the frequency approximately. In the laboratory assignment, we will see the effect of this approximation.

In the laboratory assignment, we will be developing a system that can automatically estimate the amplitude, phase, and frequency of a sinusoid. A block diagram of the "frequency, amplitude, and phase estimator" (FAPE) system is given in Figure 3.3. Unlike the APC, this system takes only two input parameters: a signal vector and the corresponding support vector. The system has four output parameters. The first three are the estimates of the frequency, amplitude, and phase of the input sinusoid. The fourth is the vector of correlations $X[1], \ldots, X[N/2]$ produced by the correlations. It is often useful to examine this vector to get a sense of what the system is doing.

Estimating doppler shift

When a sinusoidal transmitted signal reflects off an object moving towards the transmitter at speed v_o, the signal returned to the transmitter is again a sinusoid, but with the higher frequency

$$\omega_r = \omega_t * \frac{v}{v - v_o} \tag{3.54}$$

where ω_t is the original frequency of the transmitted signal and v is the speed of propagation in the given medium. This is called the *Doppler shift* phenomenon. If one measures ω_r, e.g. with FAPE, then one can use equation (3.54) to compute the speed of the object v_s, assuming of course, that ω_t and v are known. Indeed, this is how radar systems are able to measure the speed of automobiles, airplanes, baseballs, wind, etc. In the lab assignment you will be asked to estimate the velocity of an underwater object from a reflection of a sonar signal.

3.3 Some MATLAB commands for this lab

- **Constructing complex numbers:** MATLAB represents all complex numbers in rectangular form. To enter a complex number, simply type `5 + 6*j` (for instance). Note that both `i` and `j` are used to represent $\sqrt{-1}$ (unless you have used one or the other as some other variable). To enter a complex number in polar form, type `2*exp(j*pi/3)` (for instance).

  You may be wondering how MATLAB actually works with complex numbers, given that complex numbers are, in general, the sum of a real number and an imaginary one. The fact is

that the imaginary component of a complex number is in fact a *real number*, which MATLAB stores in the usual way. It thinks of a complex number as a pair of floating point numbers, one to be interpreted as the real part and the other to be interpreted as the imaginary part. And it knows the rules of arithmetic to apply to such pairs of numbers in order to do what complex arithmetic is supposed to do.

- **Extracting parts of complex numbers:** If `z` contains a complex number (or an array of complex numbers), you can find the real and imaginary parts using the commands `real(z)` and `imag(z)`, respectively. You can obtain the magnitude and angle of a complex number (or an array of complex numbers) using the commands `abs(z)` and `angle(z)`, respectively.
- **Complex conjugation:** To compute the complex conjugate of a value (or array) `z`, simply use the MATLAB command `conj(z)`.
- **Finding the index of the maximum value in a vector:** Sometimes we don't just want to find the maximum value in a vector; instead, we need to know where that maximum value is located. The `max` command will do this for us. If `v` is a vector and you use the command

  ```
  >> [max_value, index] = max(v);
  ```

  the variable `max_value` will contain largest value in the vector, and `index` contains position of `max_value` in `v`.
- MATLAB **commands to help you visually determine the amplitude, frequency, and phase of a sinusoid:** Sometimes you may need to determine the frequency, phase, and amplitude of a sinusoid from a MATLAB plot. In these cases, there three commands that are quite useful. First, the command `grid on` provides includes a reference grid on the plot; this makes it easier to see where the sinusoid crosses zero (for instance). The `zoom` command is also useful, since you can drag a zoom box to zoom in on any part of the sinusoid. Finally, you can use `axis` in conjunction with `zoom` to find the period of the signal. To do so, simply zoom in on exactly one period of the signal and type `axis`. MATLAB will return the current axis limits as `[x_min, x_max, y_min, y_max]`.
- **Calling `apc`:** The function `apc`, which you will be writing in this laboratory, estimates amplitude and phase of a continous-time target sinusoid from its samples. The input parameters are a (sampled) target sinusoid `s`, the sinusoid's support vector `t`, and the continuous-time frequency `w0` *in radians per second*. We call `apc` like this:

  ```
  >> [A,phi] = apc(s,t,w0);
  ```

  Note that a compiled version of this function, called `apc_demo.dll`, is also available.
- **Calling `fape`:** The function `fape`, which you will be writing in this laboratory, implements the frequency, amplitude, and phase estimator system. This function accepts the samples of a target continuous-time sinusoid `s` and it's support vector `t`, like this:

  ```
  >> [frq,A,phi,X] = apc(s,t);
  ```

  where `frq` is the estimated frequency in radians per second, `A` is the estimated amplitude, `phi` is the estimated phase, and `X` is the vector of correlations, $X[1], \ldots, X[N/2]$ between `s` and each reference complex exponential. Note that a compiled version of this function called, `fape_demo.dll`, is also available.

3.4 Demonstrations in the Lab Section

- Complex Numbers in MATLAB
- Sinusoids and complex exponentials in MATLAB
- Sinusoidal correlation: matching frequencies
- Sinusoidal correlation: different frequencies
- FAPE: the Frequency, Amplitude, and Phase Estimator

3.5 Laboratory Assignment

1. (Understanding sinusoids) Execute the following commands:

```
>> t = linspace(-0.5, 2, 1000);
>> plot(t,cos(linspace(-7.5,27,1000)),'k:');
```

   (a) (Extracting sinusoid parameters) Visually identify the amplitude, continuous-time frequency, and phase of the continuous-time (sampled) sinusoid that you've just plotted.
   - Include your estimated values in your report. Reduce your answers to decimal form.
   - What is the phasor that corresponds to this sinusoid? Write it in both rectangular and polar form. (Again, keep your answers in decimal form. You should use MATLAB to perform these calculations.)

   (b) (Checking your parameters) Verify your answers in the previous problem by generating a sinusoid using those parameters and plotting them on the above graph using `hold on`. Use `t` as your time axis/support vector. The new plot should be close to the original, but it does not need to be exactly correct.
   - Include the resulting graph in your report. Remember to include a `legend`.

2. (The Amplitude and Phase Calculator) In this problem we will complete and test a function which implements the "Amplitude and Phase Calculator", as described in Section 3.2.3. Download the file `apc.m`. This is a "skeleton" M-file for the "amplitude and phase calculator". Also, generate the following sinusoid (`s_test`) with its support vector (`t_test`):

```
>> t_test = 0:0.01:.99;
>> s_test = 1.3*cos(t_test*10*pi + 2.8);
```

   (a) (Identify sinusoid parameters by hand) What are the amplitude, frequency in radians per second, and phase of `s_test`?
   - Include your answers in your lab report.

   (b) (Write the APC) Complete the function `apc`. You should use the signal `s_test` to test the operation of your function. You may also wish to use the compiled function `apc_demo.dll` to test your results on other sinusoids.
   - Include the code for `apc` in your MATLAB appendix.

(c) (Test APC on a sinusoid with unknown parameters) Download the file `lab3_data.mat`. This `.mat` file contains the support vector (`t_samp`) and signal vector (`s_samp`) for a sampled continuous-time sinusoid with a continuous-time frequency of $\omega_0 = 200\pi$ radians.

- From `t_samp`, determine the sampling period, T_s, of this signal.
- Use `apc`[9] to determine the amplitude and phase of the sinusoid exactly.

(d) (APC in a non-ideal case) What happens if we use `apc` to correlate over a non-integral number of periods of our target sinusoid? We will investigate this question in this problem and the next. First, let's examine a single non-integral number of periods. Generate the following sinusoid:

```
>> apc_support = 0:0.1:8;
>> apc_test = cos(apc_support*2*pi/3);
```

This is a sinusoid with a frequency of $\omega_0 = \frac{2\pi}{3}$ radians per second, unit amplitude, and zero phase shift.

- Plot `apc_test` and include the plot in your report.
- What is the fundamental period of `apc_test`?
- Approximately how many periods are included in `apc_test`?
- Use `apc` to estimate the amplitude and phase of this sinusoid. What are the amplitude and phase errors for this signal?

(e) (APC in many non-ideal cases) Now we wish to examine a large number of different lengths of this sinusoid. You will do this by writing a `for` loop that repeats the previous part for many different values of the length of the incoming sinusoid. Specifically, write a `for` loop with loop counter `support_length` ranging over values of `1:0.1:50` seconds. In each iteration of the loop, you should

i. Set `apc_support` equal to `0:0.1:(support_length-0.1)`,
ii. Recalculate `apc_test` using the new `apc_support`,
iii. Use `apc` to estimate the amplitude and phase of `apc_test`, and
iv. Store these estimates in two separate vectors.

Put your code in an M-file script so that you can run it easily.

- Include your code in the MATLAB appendix.
- Use `subplot` to plot the amplitude and phase estimates as a function of support length in two subplots of the same figure. You should be able to see both local oscillation of the estimates and a global decrease in error with increased support length.
- At what support lengths are the amplitude estimates correct (i.e., equal to 1)?
- What minimum support length do we need to be sure that the phase error is less than 0.01 radians?

3. (The Frequency, Amplitude, and Phase Estimator) In this problem, we'll explore the frequency, amplitude and phase estimator, as described in Section 3.2.4. Download the file `fape.m`. This is a "skeleton" M-file for the "frequency, amplitude, and phase estimator" system.

[9] If you failed to correctly complete `apc.m`, you may use `apc_demo.dll` for the following problems. If you use the demo function, please indicate this in your lab report.

(a) (Write the FAPE) Complete the `fape` function. You can use `t_test` and `s_test` from Problem 2 along with the compiled `fape_demo.dll` to check your function's results.

- Include the completed code in your report's MATLAB appendix.
- What are the frequency (in radians per second), amplitude, and phase estimates returned by `fape` for `t_test` and `s_test`? Are these estimates correct?
- Use `stem` and `abs` to plot the magnitude of the vector of correlations returned by `fape` versus the associated frequencies.
- What do you notice about this plot? What can you deduce from this fact? (Hint: Consider what this plot tells you about the returned estimates.)

(b) (Running FAPE on in a non-ideal case) In this problem, we'll see what happens to FAPE when the target sinusoid does not include an integral number of periods. `lab3_data.mat` contains the variables `fape_test_t` (a support vector) and `fape_test_s` (its associated sinusoidal signal). Run `fape` on this signal.

- What are the frequency in radians per second, amplitude, and phase estimates that are returned?
- Use `stem` and `abs` to plot the magnitude of the returned vector of correlations.
- Plot `fape_test_s` and a new sinusoid that you generate from the parameter estimates returned by FAPE on the same figure (using `hold on`). Use `fape_test_t` as the support vector for the new sinusoid. Make sure you use different line types and include a legend.
- What can you say about the accuracy of estimates returned by FAPE?
- Compare the plot of the correlations generated in this problem and in Problem 3a. What do these different plots tell you?

Food for thought: Investigate the error characteristics of `fape` as you did with `apc` in problem 2e. Do the frequency, amplitude, and phase estimates improve as we use longer support lengths? Which parameter is exhibits the most error? What does the vector of correlations, $X[k]$, tell you about these estimates?

(c) Measuring speed via Doppler shift. A sonar transmitter in the ocean emits a sinusoidal signal with frequency 1000 Hz, and the signal reflects off an object moving toward the transmitter. The received signal can be found in the MATLAB workspace `lab3_data.mat`. The signal vector is called `s_sonar` and the support vector is `t_sonar`. The speed of sound in salt water is approximately 1450 meters/second. (Note: because the signal is rather long, it may take a little while for FAPE to run.)

- Estimate the speed of the object.

Food for thought: Use `randn` to add some random noise to `s_sonar` and observe how your estimate changes. How much noise do you need to add to produce an error? Does the system degrade gracefully? (That is, is the amount of error proportional to the amount of noise?)

4. On the front page of your report, please provide an estimate of the average amount of time spent outside of lab by each member of the group.

Laboratory 4

Fourier Series and the DFT

4.1 Introduction

As emphasized in the previous lab, sinusoids are an important part of signal analysis. We noted that many signals that occur in the real world are composed of sinusoids. For example, many musical signals can be approximately described as sums of sinusoids, as can some speech sounds (vowels in particular). It turns out that any periodic signal can be written exactly as a sum of amplitude-scaled and phase-shifted sinusoids. Equivalently, we can use Euler's inverse formulas to write periodic signals as sums of complex exponentials. This is a mathematically more convenient description, and the one that we will adopt in this laboratory and, indeed, in the rest of this course. The description of a signal as a sum of sinusoids or complex exponentials is known as the *spectrum* of the signal.

Why do we need another representation for a signal? Isn't the usual *time-domain* representation enough? It turns out that spectral (or *frequency-domain*) representations of signals have many important properties. First, a frequency-domain representation may be simpler than a time-domain representation, especially in cases where we cannot write an analytic expression for the signal. Second, a frequency-domain representation of a signal can often tell us things about the signal that we would not know from just the time-domain signal. Third, a signal's spectrum provides a simple way to describe the effect of certain systems (like *filters*) on that signal. There are many more uses for frequency-domain representations of a signal, and we will examine many of them throughout this course. Spectral representations are one of the most central ideas in signals and systems theory, and can also be one of the trickiest to understand.

Consider the following problem. Suppose that we have a signal that is actually the sum of two different signals. Further, suppose that we would like to separate one signal from the other, but the signals overlap in time. If the signals have frequency-domain representations that do not overlap, it is still possible to separate the two signals. In this way, we can see that frequency-domain representations provide another "dimension" to our understanding of signals.

In this laboratory, we will examine two tools that allow us to use spectral representations. The *Fourier Series* is a tool that we use to work with spectral representations of periodic continuous-time signals. The *Discrete Fourier Transform* (DFT) is an analogous tool for periodic discrete-time signals. Each of these tools allow both *analysis* (the determination of the spectrum of the time-domain signal) and *synthesis* (the reconstruction of the time-domain signal from its spectrum). Though you may not be aware of it, you have already performed DFT analysis; the "frequency, amplitude, and phase estimator" system that you implemented in Laboratory 3 actually performs DFT analysis.

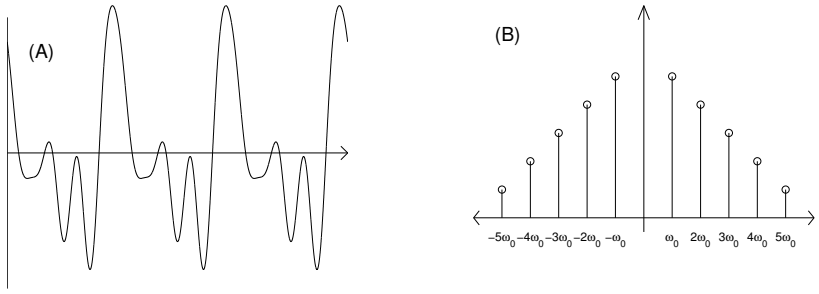


Figure 4.1: (A) A time-domain representation of a signal. (B) A frequency-domain representation of the same signal produced with the Fourier Series.

4.1.1 "The Questions"

- How can we determine the spectral content of signals?
- How can we separate signals that overlap in time?

4.2 Background

4.2.1 Frequency-domain representations

This section provides an overview to the Fourier series approach of the frequency-domain representation of continuous-time signals.

So far, we have typically thought of signals as time-varying quantities, like $s(t)$. When we plot these signals, we generally place time along the horizontal axis and signal value along the vertical axis. The idea behind the frequency-domain representation of a signal is similar. Rather than plotting signal value versus time, we plot a spectral value versus *frequency*. Doing this involves a *transformation* of the signal. Figure 4.1 shows an example of a time-domain and frequency-domain representation of a signal. Note that we can think of the result of the transform as a signal as well, a signal whose independent variable is frequency rather than time.

The frequency domain representation of a signal (i.e., its *spectrum*) is easy to construct when the signal is composed of a sum of simple complex exponential signals. In this case, the spectrum consists of a few isolated *spectral lines* ("spikes") on the frequency axis *at the frequencies of those complex exponentials*. These spectral lines are complex-valued, and their magnitudes and angles equal the amplitudes and phases of the corresponding complex exponentials. Alternatively, we may draw two separate spectral line plots — one showing the magnitude and the other showing their angles.

If we add more complex exponentials to our signal, then we simply add more spectral lines to its frequency-domain representation. Eventually, if we add enough complex exponentials (possibly an infinite number), we can create *any* signal that we might want. This includes signals that do not look very sinusoidal, like square waves and sawtooth waves. We will use this result for periodic signals in this laboratory assignment.

4.2.2 Periodic Continuous-Time Signals — The Fourier Series

Suppose that we have a periodic continuous-time signal $s(t)$ with period T seconds. We have claimed that *any* such signal can be represented as a sum of complex exponential signals. We now assert that these complex exponentials have harmonically related frequencies. Specifically, their frequencies (in radians per second) form a *harmonic series*

$$\ldots, -3\omega_0, -2\omega_0, -\omega_0, 0, \omega_0, 2\omega_0, 3\omega_0, \ldots \ , \tag{4.1}$$

where

$$\omega_0 \ = \ \frac{2\pi}{T} \tag{4.2}$$

is the *fundamental frequency*. The frequency $k\omega_0$, $k \geq 2$, is called the k-th *harmonic* of the fundamental frequency, or the k-th harmonic frequency for short.

Next we assert that the representation of $s(t)$ in terms of complex exponentials with these frequencies is given by the *Fourier Series synthesis formula*[1]:

$$\begin{aligned} s(t) &= \ldots \alpha_{-2} e^{j\frac{2\pi(-2)}{T}t} + \alpha_{-1} e^{j\frac{2\pi(-1)}{T}t} + \alpha_0 e^{j\frac{2\pi 0}{T}t} + \alpha_1 e^{j\frac{2\pi 1}{T}t} + \alpha_2 e^{j\frac{2\pi 2}{T}t} + \ldots \\ &= \sum_{k=-\infty}^{\infty} \alpha_k e^{j\frac{2\pi k}{T}t} \ , \end{aligned} \tag{4.3}$$

where the α_k's, which are called *Fourier coefficients*. The Fourier coefficients are determined by the *Fourier series analysis formula*

$$\alpha_k = \frac{1}{T} \int_{\langle T_0 \rangle} s(t) e^{-j\frac{2\pi k}{T}t} dt \ , \tag{4.4}$$

where $\int_{\langle T \rangle}$ indicates an integral over any T second interval[2]. In other words, the Fourier synthesis formula shows that the complex exponential component of $s(t)$ at frequency $\frac{2\pi k}{T}$ is

$$\alpha_k e^{j\frac{2\pi k}{T}t} \ . \tag{4.5}$$

Similarly, the Fourier analysis formula shows how the complex exponential components can be determined from $s(t)$, even when no exponential components are evident.

In general, the Fourier coefficients, i.e. the α_k's, are complex. Thus, they have a magnitude $|\alpha_k|$ and a phase or angle $\angle\alpha_k$. The magnitude $|\alpha_k|$ can be viewed as the strength of the exponential component at frequency $k\omega_0 = 2\pi k/T$, while the angle $\angle\alpha_k$ gives the phase of that component. The coefficient α_0 is the *DC term*; it measures the average value of the signal over one period.

Once we know the α_k's, the spectrum of $s(t)$ is simply a plot consisting of spectral lines at frequencies $\ldots, -2\omega_0, -\omega_0, 0, \omega_0, 2\omega_0, \ldots$. The spectral line at frequency $k\omega_0$ is drawn with height indicating the magnitude $|\alpha_k|$ and is labeled with the complex value of α_k. Alternatively, two separate spectral line plots can be drawn — one showing the $|\alpha_k|$'s and the other showing the $\angle\alpha_k$'s.

Notice that the Fourier synthesis formula is very similar to the formula given in Lab 3 for the correlation between a sinusoid and a complex exponential. Indeed it has the same interpretation: in computing α_k we are computing the correlation[3] between the signal $s(t)$ and a complex exponential with frequency $2\pi k/T$. Thought of another way, this correlation gives us an indication of *how much* of a particular complex exponential is contained in the signal $s(t)$.

[1] This is the *exponential form* of the Fourier series synthesis formula. There is also a *sinusoidal form*, which is presented later in this section.

[2] Because $s(t)e^{-j\frac{2\pi k}{T}t}$ is periodic with period T, this integral evaluates to the same value for any interval of length T.

[3] Actually, here we are computing what we called the *length-normalized correlation*.

Partial Series

Notice the infinite limits of summation in the synthesis formula (4.3). This tells us that, for the general case, we need an infinite number of complex exponentials to represent our signal. However, in practical situations, such as in this lab assignment, when we use the synthesis formula to determine signal values, we can generally only include a finite number of terms in the sum. For example, if we use only the first N positive and negative frequencies plus the DC term (at $k = 0$), our approximate synthesis equation becomes

$$s(t) \approx \sum_{k=-N}^{N} \alpha_k e^{j\frac{2\pi k}{T}t} \; . \tag{4.6}$$

Fortunately, Fourier series theory shows that this approximation becomes better and better[4] as $N \longrightarrow \infty$. Alternatively, it is known that the mean-squared value of the difference between $s(t)$ and the approximation tends to zero as $N \longrightarrow \infty$. Specifically, it can be shown that

$$\begin{aligned} MS\left(s(t) - \sum_{k=-N}^{N} \alpha_k e^{j\frac{2\pi k}{T}t}\right) &= MS(s(t)) - \sum_{k=-N}^{N} |\alpha_k|^2 \\ &\longrightarrow 0 \text{ as } N \longrightarrow \infty \; . \end{aligned} \tag{4.7}$$

How large must N be for the approximation to be good? There is no simple answer. However, you will gain some idea by the experiments you perform in this lab assignment.

T-Second Fourier Series

If a signal $s(t)$ is periodic with period T, then it is also periodic with period $2T$, and period $3T$, and so on. Thus when applying Fourier series, we have a choice as to the value of T. Often, we will choose T to be the smallest period, i.e. the *fundamental period* of $s(t)$. However, there are also situations where we will not. For example, suppose we wish to perform spectral analysis/synthesis of two or more periodic signals that have different fundamental periods. We could of course form a separate Fourier series for each signal. In this case, each Fourier series will be based on a different harmonic series of frequencies. Wouldn't it be nicer if we could base each series on a common harmonic series of frequencies? We can do this by choosing T to be a multiple of the fundamental periods of both signals.

When we want to explicitly specify the value of T that is used in a Fourier series, we will say *T-second Fourier series*. What then is the relationship between Fourier series corresponding to different values of T? To see what is happening, let us compare a T-second Fourier series to a $2T$-second Fourier series. The T-second Fourier series has components at the frequencies

$$\ldots, -2\omega_0, -\omega_0, 0, \omega_0, 2\omega_0, \ldots \; , \tag{4.8}$$

where

$$\omega_0 = \frac{2\pi}{T} \tag{4.9}$$

and the $2T$-second Fourier series has components at the frequencies.

$$\ldots, -2\omega_0', -\omega_0', 0, \omega_0', 2\omega_0', \ldots \;=\; \ldots, -\omega_0, -\frac{\omega_0}{2}, 0, \frac{\omega_0}{2}, \omega_0, \ldots \; , \tag{4.10}$$

[4] It is known that under rather benign assumptions about the signal $s(t)$, the approximation converges to $s(t)$ as $N \longrightarrow \infty$ at all times t where $s(t)$ is continuous, and at times t where $s(t)$ has a jump discontinuity, the approximation converges to the average of the values immediately to the left and right of the discontinuity.

where

$$\omega_0' = \frac{2\pi}{2T} = \frac{\omega_0}{2}. \tag{4.11}$$

From this we see that the $2T$-second Fourier series decomposes $s(t)$ into frequency components with half the separation of that of the T-second Fourier series. However, since $s(t)$ is periodic with period T, its spectrum is actually concentrated at frequencies that are multiples of ω_0 (or a subset thereof). Hence, the "additional" coefficients in the $2T$-Fourier series must be zero, and it turns out that the nonzero coefficients are the same as for the T-second Fourier series. Specifically, it can be shown that with α_k and α_k' denoting the T-second and $2T$-second Fourier coefficients, respectively, then

$$\alpha_k' = \begin{cases} \alpha_{k/2}, & k \text{ even} \\ 0, & k \text{ odd} \end{cases} \tag{4.12}$$

In summary, Fourier series analysis/synthesis can be performed over one fundamental period or over any number of fundamental periods. Usually, when Fourier series is mentioned, the desired number of periods interval will be clear from context. In any case, the spectrum is not affected by the choice of T.

Aperiodic Continuous-Time Signals

Next, we briefly discuss how Fourier series can also be applied when the signal $s(t)$ is not periodic. In this case, we can nevertheless determine the spectrum of a finite *segment* of the signal, say from time t_1 to time t_2, by performing Fourier series analysis/synthesis on just this segment. That is, if we find Fourier coefficients

$$\alpha_k = \frac{1}{T}\int_{t_1}^{t_2} s(t)e^{-j\frac{2\pi k}{T}t}dt\ , \tag{4.13}$$

where $T = t_2 - t_1$, then we have

$$s(t) = \sum_{k=-\infty}^{\infty} \alpha_k e^{j\frac{2\pi k}{T}t}\ , \text{ for } t_1 \le t \le t_2\ . \tag{4.14}$$

This will give us an idea of the frequency content of the signal during the given time interval. It is important to emphasize, however, that the synthesis equation (4.14) is valid *only* when t is between t_1 and t_2. Outside of this time interval, the synthesis formula will not necessarily equal $s(t)$. Instead, it describes a signal that is periodic with period T, called the *periodic extension* of the segment between t_1 and t_2.

Properties of the Fourier Coefficients

We conclude our discussion of the Fourier series with a list of useful properties, some of which have already been mentioned. A few of these will be useful in this lab assignment. The rest are included for completeness. These properties are stated without derivations. However, each can be derived straightforwardly from the analysis and synthesis formulas. Though not required in this laboratory, you may want to confirm some of these properties using the Fourier analysis and synthesis programs described in Section 4.3.

1. (Fourier series analysis) The T-second Fourier series analysis of a periodic signal $s(t)$ with period T produces a set of Fourier coefficients α_k, $k = \ldots, -2, -1, 0, 0, 1, 2, \ldots$, which are, in general, complex valued.

2. (Frequency components) If α_k are the coefficients of the T-second Fourier series of the periodic signal $s(t)$ with period T, then the frequency or spectral component of $s(t)$ at frequency $\frac{2\pi k}{T}$ is $\alpha_k e^{j\frac{2\pi k}{T}t}$.

3. (DC component) The coefficient α_0 equals the average or DC value of $s(t)$.

4. (One-to-one relationship) There is a one-to-one relationship between periodic signals and Fourier coefficients. Specifically, if $s(t)$ and $s'(t)$ are distinct[5] periodic signals, each periodic with period T, then their T-second Fourier coefficients are not entirely identical, i.e. $\alpha_k \neq \alpha'_k$ for at least one k. It follows that one can recognize a periodic signal from its Fourier coefficients (and its period).

5. (Conjugate symmetry) If $s(t)$ is a real-valued signal, i.e. its imaginary part is zero, then for any integer k

$$\begin{aligned} \alpha_{-k} &= \alpha_k^* && (4.15)\\ |\alpha_{-k}| &= |\alpha_k| && (4.16)\\ \angle\alpha_{-k} &= -\angle\alpha_k\ . && (4.17) \end{aligned}$$

6. (Conjugate pairs) If α_k's are the T-second Fourier coefficients for a real-valued signal $s(t)$, then for any k the sum of the complex exponential components of $s(t)$ corresponding to α_k and α_{-k} is a sinusoid at frequency $2\pi k/T$. Specifically, using the inverse Euler relation,

$$\alpha_k e^{j\frac{2\pi k}{T}t} + \alpha_{-k} e^{-j\frac{2\pi k}{T}t} = 2|\alpha_k|\cos(\frac{2\pi k}{T}t + \angle\alpha_k)\ . \tag{4.18}$$

7. (Sinusoidal form of the Fourier synthesis formula) The previous property leads to the sinusoidal form of the Fourier synthesis formula:

$$s(t) = \alpha_0 + \sum_{k=-\infty}^{\infty} 2|\alpha_k|\cos(\frac{2\pi k}{T}t + \angle\alpha_k)\ . \tag{4.19}$$

8. (Linear combinations) If $s(t)$ and $s'(t)$ have T-second Fourier coefficients α_k and α'_k, respectively, then $as(t) + bs'(t)$ has T-second Fourier coefficients $a\alpha_k + b\alpha'_k$.

9. (Fourier series of elementary signals) The following lists the T-second Fourier coefficients of some elementary signals.

   (a) Complex exponential signal: $s(t) = e^{j\frac{2\pi m}{T}t} \implies$

$$\alpha_k = \begin{cases} 1, & k = m \\ 0, & k \neq m \end{cases}\ . \tag{4.20}$$

[5] By "distinct", we mean that $s(t)$ and $s'(t)$ are sufficiently different that $s(t) \neq s'(t)$ for all times t in some interval with (t_1, t_2), with nonzero length. They are *not* "distinct" if they differ only at a set of isolated points. To see why we need this clarification, observe that if $s(t)$ and $s'(t)$ differ only at time t_1, then they have the same Fourier coefficients, because integrals, such as those defining Fourier coefficients, are not affected by changes to their integrands at isolated points. Likewise, $s(t)$ and $s'(t)$ will have the same Fourier coefficients if they differ only at isolated times $t_1, t_2, \ldots$. However, if $s(t) \neq s'(t)$ for all t in an entire interval, no matter how small, then $\alpha_k \neq \alpha'_k$ for at least one k.

(b) Cosine: $s(t) = \cos(\frac{2\pi m}{T}t) \implies$

$$\alpha_k = \begin{cases} \frac{1}{2}, & k = \pm m \\ 0, & k \neq \pm m \end{cases} . \tag{4.21}$$

(c) Sine: $s(t) = \sin(\frac{2\pi m}{T}t) \implies$

$$\alpha_k = \begin{cases} -\frac{j}{2}, & k = m \\ \frac{j}{2}, & k = -m \\ 0, & k \neq \pm m \end{cases} . \tag{4.22}$$

(d) General sinusoid: $s(t) = \cos(\frac{2\pi m}{T}t + \phi) \implies$

$$\alpha_k = \begin{cases} \frac{1}{2}e^{j\phi}, & k = m \\ \frac{1}{2}e^{-j\phi}, & k = -m \\ 0, & k \neq \pm m \end{cases} . \tag{4.23}$$

10. (T-second Fourier series) If a periodic signal $s(t)$ has period T and T-second Fourier coefficients α_k, then the nT-second Fourier coefficients are

$$\alpha'_k = \begin{cases} \alpha_{k/n}, & k = \text{ multiple of } n \\ 0, & \text{else} \end{cases} \tag{4.24}$$

11. (Parseval's relation) If α_k's are the T-second Fourier coefficients for signal $s(t)$, then the mean-squared value of $s(t)$, equivalently the power, equals the sum of the squared magnitudes of the Fourier coefficients. That is,

$$MS(s) = \frac{1}{T}\int_{\langle T \rangle} |s(t)|^2 \, dt = \sum_{k=-\infty}^{\infty} |\alpha_k|^2 \tag{4.25}$$

12. (Uncorrelatedness/orthogonality of complex exponentials) The T-second correlation between complex exponential signals $e^{j\frac{2\pi m}{T}t}$ and $e^{j\frac{2\pi n}{T}t}$, $m \neq n$, is zero. This property is used in the derivation of the previous and other properties.

4.2.3 Periodic Discrete-Time Signals — The Discrete Fourier Transform

This section overview the discrete Fourier transform approach to the frequency-domain representation of discrete-time signals.

Consider a periodic discrete-time signal $s[n]$ with period N. As with continuous-time signals, we wish to find its frequency-domain representation, i.e. its spectrum. That is, we wish to represent $s[n]$ as a sum of *discrete-time* complex exponential signals. Again, by analogy to the continuous-time case we will use frequencies that are multiples of

$$\widehat{\omega}_0 = \frac{2\pi}{N} . \tag{4.26}$$

However, unlike the continuous-time case, we now use only a finite number of such frequencies. Specifically, we use the N harmonically related frequencies:

$$0, \widehat{\omega}_0, 2\widehat{\omega}_0, \ldots, (N-1)\widehat{\omega}_0 . \tag{4.27}$$

The reason is that any complex exponential signal with the frequency $k\widehat{\omega}_0$ is in fact identical to a complex exponential signal with one of the N frequencies listed above[6]. Notice that this set of frequencies ranges from 0 to $\frac{2\pi(N-1)}{N}$, which is just a little less than 2π.

We now assert that the representation of $s[n]$ in terms of complex exponentials with the above frequencies is given by the *discrete-time Fourier series synthesis formula* or as we will usually call it, the *the Discrete Fourier Transform (DFT) synthesis formula*

$$\begin{aligned} s[n] &= S[0]e^{j\frac{2\pi 0}{N}n} + S[1]e^{j\frac{2\pi 1}{N}n} + S[2]e^{j\frac{2\pi 2}{N}n} + \ldots + S[N-1]e^{j\frac{2\pi(N-1)}{N}n} \\ &= \sum_{k=0}^{N-1} S[k]e^{j\frac{2\pi k}{N}n}\,, \end{aligned} \tag{4.28}$$

where the $S[k]$'s, which are called *DFT coefficients*, are determined by the *DFT analysis formula*

$$S[k] = \frac{1}{N}\sum_{\langle N\rangle} s[n]e^{-j\frac{2\pi k}{N}n}\,, \qquad k = 0, 1, 2, 3, \ldots, N-1 \tag{4.29}$$

where $\langle N\rangle$ indicates a sum over any N consecutive integers[7], e.g. the sum over $0, \ldots, N$.

As with the continuous-time Fourier series, the DFT coefficients are, in general, complex. Thus, they have a magnitude $|S[k]|$ and a phase or angle $\angle S[k]$. The magnitude $|S[k]|$ can be viewed as the strength of the exponential component at frequency $k\widehat{\omega}_0 = 2\pi k/N$, while $\angle S[k]$ is the phase of that component. The coefficient $S[0]$ is the *DC term*; it measures the average value of the signal over one period.

Once we know the $S[k]$'s, the spectrum of $s[n]$ is simply a plot consisting of spectral lines at frequencies $0, \widehat{\omega}_0, 2\widehat{\omega}_0, \ldots, (N-1)\widehat{\omega}_0$. The spectral line at frequency $k\widehat{\omega}_0$ is drawn with height indicating the magnitude $|S[k]|$ and is labeled with the complex value of $S[k]$. Alternatively, two separate spectral line plots can be drawn — one showing the $|S[k]|$'s and the other showing the $\angle S[k]$'s.

Since the sums in the synthesis and analysis formulas are finite, there are no convergence-of-partial-sum issues, such as those that arise for the continuous-time Fourier series.

Often the DFT coefficients $S[0], \ldots, S[N]$ are said to be the "DFT of the signal $s[n]$" and the process of computing them via the analysis equation (4.29) is called "taking the DFT" of $s[n]$. Conversely, applying the synthesis equation (4.28) is often called "taking the inverse DFT" of $S[0], \ldots, S[N]$.

Notice that the DFT analysis formula (4.29) is identical to equation (3.45) in Lab 3. That is, in computing the set of correlations between a signal $s[n]$ and the various complex exponentials in Lab 3, we were actually taking the DFT of $s[n]$. Indeed, it continues to be helpful to view the DFT analysis as the process of correlating $s[n]$ with various complex exponentials. Those correlations that lead to larger magnitude coefficients indicate frequencies where the signal has larger components.

In some treatments, the DFT analysis and synthesis formulas differ slightly from those given above in that the $1/N$ factor is moved from the analysis formula to the synthesis formula[8], or replaced by a $1/\sqrt{N}$ factor multiplying each formula. All of these approaches are equally valid. The choice between them is largely a matter of taste. For example, our approach is the only one for

[6] If $k\widehat{\omega}_0$ is not in this range, then $k = mN + l$ where $m \neq 0$ and $0 \leq l < N$. It then follows that the complex exponential with this frequency is $e^{j\frac{2\pi k}{N}n} = e^{j\frac{2\pi(mN+l)}{N}n} = e^{j2\pi mn}e^{j\frac{2\pi l}{N}n} = e^{j\frac{2\pi l}{N}n}$, which is an exponential with one of the N frequencies in the list above.

[7] Because $s[n]e^{-j\frac{2\pi k}{N}n}$ is periodic with period N, the sum is the same for any choice of N consecutive integers.

[8] The *DSP First* textbook does this in Chapter 9.

which $S[0]$ equals the average signal value. For the other approaches, the average is $S[0]$ multiplied by a known constant. The only cautionary note is that one should never use the analysis formula from one version with the synthesis formula from another. In this course, we will always use the analysis and synthesis formulas shown above.

Although we will always take $0, \widehat{\omega}_0, 2\widehat{\omega}_0, \ldots, (N-1)\widehat{\omega}_0$ as the analysis frequencies produced by the DFT, it is important to point out that every frequency $\widehat{\omega}$ in the upper half of this range, i.e. between π and 2π, is equivalent to a frequency $\widehat{\omega} - 2\pi$, which lies between $-\pi$ and 0. By "equivalent," we mean that a complex exponential with frequency $\widehat{\omega}$ with $\pi < \widehat{\omega} < 2\pi$ equals the complex exponential with frequency $\widehat{\omega} - 2\pi$. Thus, it is often useful to think of frequencies in the upper half of our designated range as representing frequencies in the range $-\pi$ to 0.

For example, let us look at the DFT of a sinusoidal signal, $s[n] = \cos(\frac{2\pi m}{N} n)$, with $0 < m < \frac{N}{2}$. The DFT coefficients, $S[k]$, are given by

$$(S[0], \ldots, S[N-1]) = (0, \ldots, 0, 1/2, 0, \ldots, 0, 1/2, 0, \ldots, 0), \tag{4.30}$$

where $S[m] = S[N-m] = 1/2$ and $S[k] = 0$ for other k's. In the synthesis formula, the coefficient $S[m]$ multiplies the complex exponential $e^{j\frac{2\pi m}{N}n}$, and the coefficient $S[N-m]$ multiplies the complex exponential $e^{j\frac{2\pi(N-m)}{N}n} = e^{-j\frac{2\pi m}{N}n}$. Thus, these two coefficients can be viewed as multiplying exponentials at frequencies $\pm\frac{2\pi m}{N}$, which by the inverse Euler formula sum to yield $s[n] = \cos(\frac{2\pi m}{N} n)$.

N-point DFT

As with continuous-time signals, if a discrete-time signal $s[n]$ is periodic with period N, then it also periodic with period $2N$, and period $3N$, and so on. Thus, when applying the DFT, we have a choice as to the value of N. Sometimes we choose it to be the the smallest period, i.e. the fundamental period, but sometimes we do not. When we want to explicitly specify the value of N used in a DFT, we will say *N-point DFT*.

The relationship between the N-point and $2N$-point DFT is just like the relationship between the T-second and $2T$-second Fourier series. That is, whereas the N-point DFT has components at frequencies

$$0, \widehat{\omega}_0, 2\widehat{\omega}_0, \ldots, (N-1)\widehat{\omega}_0\ , \tag{4.31}$$

the $2N$-point DFT has components at the frequencies

$$0, \widehat{\omega\prime}_0, 2\widehat{\omega}'_0, \ldots, (2N-1)\widehat{\omega}'_0 \ = \ 0, \frac{\widehat{\omega}_0}{2}, \widehat{\omega}_0, \ldots, (2N-1)\frac{\widehat{\omega}_0}{2}\ . . \tag{4.32}$$

where

$$\widehat{\omega}_0 \ = \ \frac{2\pi}{2N} \ = \ \frac{\omega_0}{2} \tag{4.33}$$

From this we see that the separation between frequency components has been halved. Moreover, it can be shown that the relationship between the original and new coefficients is

$$S'[k] = \begin{cases} S[k/2], & k \text{ even} \\ 0, & k \text{ odd} \end{cases} \tag{4.34}$$

In summary, DFT analysis/synthesis can be performed over one fundamental period or over any number of fundamental periods. Usually, when the DFT is mentioned, the desired number of periods interval will be clear from context. In any case, the spectrum is not affected by the choice of N.

Aperiodic Discrete-Time Signals

Next, we briefly discuss how the DFT can also be applied when the signal $s[n]$ is not periodic. In this case, we can nevertheless determine the spectrum of a finite *segment* of the signal, say from time n_1 to time n_2, by performing DFT analysis/synthesis on just this segment. That is, if we find DFT coefficients

$$S'[k] = \frac{1}{N} \sum_{\langle N \rangle} s[n] e^{-j\frac{2\pi k}{N}n} , \qquad k = 0, 1, 2, 3, \ldots, N-1 \tag{4.35}$$

where $N = n_2 - n_1$, then we have

$$s[n] = \sum_{k=0}^{N-1} S'[k] e^{j\frac{2\pi k}{N}n} , \qquad k = 0, 1, 2, 3, \ldots, N-1 . \tag{4.36}$$

This will give us an idea of the frequency content of the signal during the given time interval. It is important to emphasize, however, that the synthesis equation (4.36) is valid *only* at times n from n_1 to n_2. Outside of this time interval, the synthesis formula will not necessarily equal $s[n]$. Instead, it describes a signal that is periodic with period N, called the *periodic extension* of the segment from n_1 to n_2.

Approximating Fourier series coefficients with the DFT

Frequently, we are interested in finding the spectrum of some continuous-time signal $s(t)$, but for practical reasons, we sample the signal and work with the resulting discrete-time signal $s[n]]$. Can we find, at least approximately, the spectrum of $s(t)$ by working with the discrete-time signal $s[n]$? As discussed below there is a close relationship between the Fourier series coefficients of $s(t)$ and the DFT of $s[n]$.

Suppose $s(t)$ is periodic with period T, and suppose we sample $s(t)$ with sampling interval $T_s = T/N$, where N is an integer, resulting in the discrete-time signal $s[n] = s(nT_s)$, which is easily seen to be periodic with period N. Let α_k denote the T-second Fourier coefficients of $s(t)$, and let $S[k]$ denote the N-point DFT of $s[n]$. Then it can be shown that if N is very large, then

$$\alpha_k \approx S[k] , \text{ when } k << N \tag{4.37}$$

Moreover, it can be shown that if it should happen that $s(t)$ has no spectral components at frequencies greater than $1/(2T_s)$, then

$$\alpha_k = \begin{cases} S[k], & 0 \le k \le N/2 \\ S[N-k+1], & -N/2 \le k < 0 \\ 0, & |k| > N/2 \end{cases} \tag{4.38}$$

The above two equation shows how the DFT can be used to compute, at least approximately, the Fourier series coefficients. In fact, the Fourier series analysis program described in the in the MATLAB section of this assignment uses the DFT to compute the Fourier coefficients.

Properties of the DFT coefficients

The following are a number of useful properties of the DFT with which you should be familiar. A few of these will be useful in this lab assignment. Others will be used in future assignments. These

properties are stated without derivations. However, each can be derived straightforwardly from the analysis and synthesis formulas. Though not required in this laboratory, you may want to confirm some of these properties using the DFT analysis and synthesis programs described in Section 4.3.

1. (DFT analysis) The N-point DFT of a periodic signal $s[n]$ with period N produces a vector of N DFT coefficients $S[0], \ldots, S[N-1]$, which are, in general, complex valued. Equivalently, the coefficients may be considered to be determined by a set of N signal samples.

2. (Frequency components) If $S[k]$ is N-point DFT of the periodic signal $s[n]$ with period N, then the frequency or spectral component of $s[n]$ at frequency $\frac{2\pi k}{N}$ is $S[k]e^{j\frac{2\pi k}{N}n}$. The component of the signal at frequency $\frac{-2\pi k}{N}$ is $S[N-k]e^{-j\frac{2\pi k}{N}N}$.

3. (DC component) The coefficient $S[0]$ equals the average value or DC value of $s[n]$.

4. (One-to-one relationship) There is a one-to-one relationship between discrete-time signals with period N (equivalently, sequences of N signal samples) and sequences of N DFT coefficients. Specifically, if $s[n]$ and $s'[n]$ are distinct periodic signals with period N, i.e. $s[n] \neq s'[n]$ for some value of n, then their N-point DFT coefficients are not entirely identical, i.e. $S[k] \neq S'[k]$ for at least one k. It follows that one can recognize a discrete-time periodic signal from its DFT coefficients (and N).

5. (Conjugate symmetry) If $s[n]$ is a real-valued signal, i.e. its imaginary part is zero, then for any integer k

$$\begin{aligned} S[N-k] &= S^*[k] && (4.39)\\ |S[N-k]| &= |S[k]| && (4.40)\\ \angle S[N-k] &= -\angle S[k]\ . && (4.41) \end{aligned}$$

   These facts indicate that we are usually only interested in the first half of the DFT coefficients. In particular, note that when we plot the DFT, the location of the origin and the appearance of the symmetry is different than when we plot the Fourier Series. See Figure 4.2 for an example of the relation between the two.

6. (Conjugate pairs) If $S[k]$ is the N-point DFT of a real-valued signal $s[n]$, then for any k the sum of the complex exponential components of $s[n]$ corresponding to $S[k]$ and $S[N-k]$ is a sinusoid at frequency $2\pi k/N$. Specifically, using the inverse Euler relation,

$$S[k]e^{j\frac{2\pi k}{N}n} + S[N-k]e^{-j\frac{2\pi k}{N}n} = 2|S[k]|\cos(\frac{2\pi k}{N}n + \angle S[k])\ . \qquad (4.42)$$

7. (Linear combinations) If $s[n]$ and $s'[n]$ have N-point DFT $S[k]$ and $S'[k]$, respectively, then $as[n] + bs'[n]$ has N-point DFT $aS[k] + bS'[k]$.

8. (Sampled continuous-time signals) If the discrete-time signal $s[n]$ comes from sampling a continuous-time signal $s(t)$ with sampling interval T_s, i.e. if $s[n] = s(nT_s)$, then the continuous-time frequency represented by DFT coefficient $S[k]$ is $\frac{2\pi k}{N}f_s$, where $f_s = 1/T_s$ samples per second is the sampling rate.

9. (DFT of elementary signals) The following lists the N-point DFT of some elementary signals.

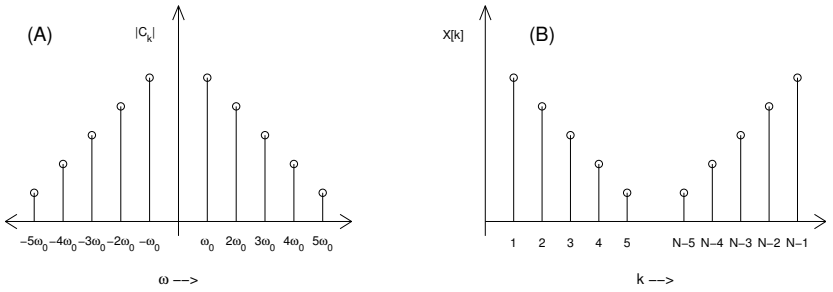


Figure 4.2: (A) The magnitude of the Fourier Series coefficients α_k for a periodic continuous-time signal. (B) The DFT of a periodic discrete-time version of the same signal. Note that the origin for the Fourier Series coefficients is in the middle of the plot, but the origin for the DFT is to the left.

(a) Complex exponential signal: $s[n] = e^{j\frac{2\pi m}{N}n} \implies$

$$(S[0], \ldots, S[N-1]) = (0, \ldots, 0, 1, 0, \ldots, 0) \,, \tag{4.43}$$

where the nonzero coefficient is $S[m]$.

(b) Cosine: $s[n] = \cos\left(\frac{2\pi m}{N}n\right) \implies$

$$(S[0], \ldots, S[N-1]) = (0, \ldots, 0, \frac{1}{2}, 0, \ldots, 0, \frac{1}{2}, 0, \ldots, 0) \,, \tag{4.44}$$

where the nonzero coefficients are $S[m]$ and $S[N-m]$.

(c) Sine: $s[n] = \sin\left(\frac{2\pi m}{N}n\right) \implies$

$$(S[0], \ldots, S[N-1]) = (0, \ldots, 0, -\frac{j}{2}, 0, \ldots, 0, \frac{j}{2}, 0, \ldots, 0) \,, \tag{4.45}$$

where the nonzero coefficients are $S[m]$ and $S[N-m]$.

(d) General sinusoid: $s[n] = \cos\left(\frac{2\pi m}{N}n + \phi\right) \implies$

$$(S[0], \ldots, S[N-1]) = (0, \ldots, 0, \frac{1}{2}e^{j\phi}, 0, \ldots, 0, \frac{1}{2}e^{-j\phi}, 0, \ldots, 0) \,, \tag{4.46}$$

where the nonzero coefficients are $S[m]$ and $S[N-m]$.

(e) Not quite periodic sinusoid: $s[n] = \cos\left(\frac{2\pi(m+\epsilon)}{N}n\right)$ where $(m+\epsilon)$ is non-integer $\implies$ The resulting $S[k]$'s will all be nonzero[9], typically with small magnitudes except those corresponding to frequencies closest to $\frac{2\pi(m+\epsilon)}{N}$.

(f) Period contains unit impulse period: $s[n] = (1, 0, \ldots, 0) \implies$

$$(S[0], \ldots, S[N-1]) = \left(\frac{1}{N}, \ldots, \frac{1}{N}\right) \,. \tag{4.47}$$

[9]This is the same effect that you saw in lab 3 when you ran `fape` over a non-integer number of periods of the sinusoid.

10. (N-point DFT) If $S[k]$ is the N-point DFT of the periodic signal $s[n]$ with period N, then the mN-point DFT coefficients are

$$S[k] = \begin{cases} S[k/m], & k = \text{ multiple of } m \\ 0, & \text{else} \end{cases} \tag{4.48}$$

11. (Parseval's relation) If $S[k]$ is the N-point DFT of $s[n]$, then

$$MS(x) = \frac{1}{N} \sum_{\langle N \rangle} |s[n]|^2 = \sum_{k=0}^{N-1} |S[k]|^2 \, . \tag{4.49}$$

    This shows that the power in the signal $s[n]$ equals the energy of the DFT coefficients.

12. (Uncorrelatedness/orthogonality of complex exponentials) The N-point correlation between complex exponential signals $e^{j\frac{2\pi m}{N}n}$ and $e^{j\frac{2\pi l}{N}n}$, $m \neq l$, is zero. This property is used in the derivation of the previous one.

4.2.4 Separating Signals Based on Differing Harmonic Series

We've already suggested that there are many nearly-periodic signals that occur in the real world, with two notable examples being many musical signals and vowels in speech signals. These sort of signals can be analyzed using the Fourier Series or the DFT (applied to samples). We will use the DFT, principally because if we wanted to use the Fourier series, we would anyway approximately compute the Fourier coefficients with the DFT. In particular, let us consider a note played on a musical instrument like a flute or clarinet. Such a signal is nearly periodic with some fundamental period. If the note is played at "concert pitch," for instance, it has a fundamental frequency of 440 Hz and a fundamental period of $1/440$ seconds. Few musical signals, though, are purely sinusoidal. From our development of the Fourier series, we know that a periodic signal can be described as a sum of complex exponentials (or sinusoids) with harmonically-related frequencies. That is, the spectrum of our musical note is composed of a *harmonic series*. In particular, if the fundamental frequency is 440 Hz, higher harmonics will be at 880 Hz, 1320 Hz, 1760 Hz, and so on. Figure 4.3 shows a stemp plot of the DFT of an example harmonic series.

Suppose that we have two instruments playing different notes (i.e., the two signals have different fundamental periods) at the same time. The signal coming from each instrument is a single harmonic

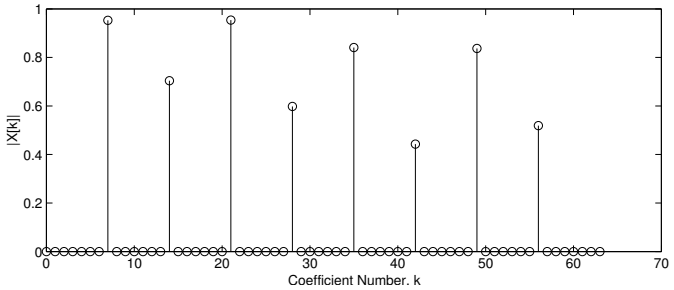


Figure 4.3: The DFT of a harmonic series. Note that only the first half of the DFT coefficients are shown in this figure.

series, but a listener "hears" a signal which is the sum of these two signals. By the linear combination properties of the Fourier Series and DFT, we know that the spectrum of the combined signal is simply the sum of the spectra of the separate signals. We can use this property to separate the two signals in the frequency-domain, even though they overlap in the time-domain.

Suppose that we wish to simply remove one of the notes from the combined signal. We'll assume that we have recorded and sampled the signal, so we're working in discrete-time. We'll also assume that the combined signal is also periodic[10] with some (fairly long) fundamental period N_0. If we take the N_0-point DFT of a segment of the combined signal, we can identify the coefficients that make up each harmonic series. Then, we simply zero-out all of the coefficients corresponding to the harmonics of the note we wish to remove. When we resynthesize the signal with the inverse DFT, the resulting signal will contain only one of the two notes.

We can extend this procedure to more complicated signals, like melodies with many notes. In this case, we simply analyze and resynthesize each note individually. Of course, with more simultaneously-sounding notes and more complicated music, this procedure becomes rather difficult. In this lab, we will implement this procedure to remove a "corrupting" note held throughout a simple, easily analyzed melody. Though somewhat idealized, the problem should help to motivate the use of the DFT and the frequency domain.

4.3 Some MATLAB commands for this lab

- **Fourier Series Synthesis in MATLAB:** The function `fourier_synthesis` is a function that we provide to compute the approximate T-second Fourier series synthesis formula, equation (4.6). Its inputs are the period T and a set of $2N+1$ Fourier coefficients. Its output is the synthesized signal. The calling command is

  ```
  >>   [ss,tt] = fourier_synthesis(CC, T, periods, Ns);
  ```

  where `CC` is a vector containing the Fourier coefficients, `T` is the interval (in seconds) over which the Fourier series is applied. `periods` is the (integer) number of periods to include in the resynthesis; `periods` defaults to a value of 1 if not provided. The optional parameter `Ns` specifies how many samples per period to include in the output signal.

  It is assumed that `CC` contains the coefficients $\alpha_{-N} \ldots \alpha_N$. ($N$ is implicitly determined from the length of `CC`.) Thus, `CC` has length $2N+1$, the `CC(n)` element contains the Fourier series coefficient α_{n-N-1}. Further, note that the α_0 coefficient falls at `CC(N+1)`.

  The two returned parameters are the signal vector `ss` and the corresponding signal support vector `tt`.

- **Fourier Series Analysis in MATLAB:** The function `fourier_analysis` is the complement to `fourier_synthesis`. It performs T-second Fourier series analysis on an input signal. The calling command is

  ```
  >> [CC,ww] = fourier_analysis(ss,T,N);
  ```

[10] In the "real-world," this is a somewhat questionable assumption. However, we can approximate this behavior quite well by simply using a long DFT. In this case, each harmonic may be "spread" over several DFT coefficients, so to remove a harmonic we need to zero-out all of coefficients associated with it. This spreading behavior is the same as what you saw in Lab 3 when running `fape` over non-periodic signals.

where `ss` is a vector containing the signal samples, `T` is the interval T in seconds over which the Fourier series is to be computed, and `N` is the number of positive harmonics to include in the analysis. (`2N+1` is the total number of harmonics.) It is assumed that `ss` contains samples of the signal to be analyzed over the interval $[0, T]$.

The outputs are the vectors `CC`, which contains the $2N + 1$ Fourier coefficients[11], and `ww`, which contains the frequencies (in Hertz) associated with each Fourier coefficient.

- **DFT Analysis in MATLAB:** In order to calculate an N-point DFT using MATLAB, we use the `fft` command[12]. The specific calling command is

```
>>    XX = fft(xx)/length(xx);
```

This computes the N-point DFT of the signal vector `xx` , where N is the length of `xx`, and where the signal is assumed to have support $0, 1, \ldots, N-1$. Since the MATLAB command `fft` does not include the factor $1/N$ in the analysis formula, as in equation (4.29), we must divide by `length(xx)` to obtain the N DFT coefficients `XX`.

- **DFT Synthesis in MATLAB:** The synthesis equation for the DFT is computed with the command `ifft`. If we have computed the DFT using the above command, we must also remember to multiply the result by N:

```
>> xx = ifft(XX)*length(XX);
```

Note that the `ifft` command will generally return complex values even when the synthesis should exactly be real. However, the imaginary part should be negligible (i.e., less than 1×10^{-14}). You can eliminate this imaginary part using the `real` command.

- **Indexing the DFT:** Since MATLAB begins its indexing from 1 rather than 0, remember to use the following rules for indexing the DFT:

$$\begin{aligned} X[0] &\Rightarrow \texttt{X(1)} \\ X[1] &\Rightarrow \texttt{X(2)} \\ X[k] &\Rightarrow \texttt{X(k+1)} \\ X[N-k] &\Rightarrow \texttt{X(N-k+1)} \\ X[N-1] &\Rightarrow \texttt{X(N)} \end{aligned}$$

4.4 Demonstrations in the Lab Section

- Approximating signals as sums of sinusoids, as in Problem 1.
- "Mapping out" this week's background section
- Relating the Fourier Series to the DFT
- T-second Fourier Series and the N-point DFT
- The DFT in MATLAB

[11] Because `fourier_analysis` is given only samples of the desired continuous-time signal, it cannot compute the Fourier coefficients exactly. Rather it computes an approximation by using the DFT.

[12] FFT stands for the *Fast Fourier Transform*, which is a fast implementation of the DFT. Calculating the DFT from its definition requires $O(N^2)$ computations, but the FFT only requires $O(N \log N)$. Additionally, the FFT is faster when N is equal to a power of two (i.e., $N = 256, 512, 1024, 2048$, etc.).

4.5 Laboratory Assignment

1. (Building signals from sinusoids) In this problem, you will "hand tune" the amplitudes and phases of three sinusoids so that their sum matches a "target" periodic signal as well as possible. The signals are considered to be continuous-time. One could do this task analytically or numerically using the Fourier series analysis formula, but we want you to gain the insight that results from doing it manually. A graphical MATLAB program has been written to facilitate this procedure.

   Download the files `sinsum.m` and `sinsum.fig` and execute `sinsum`[13]. MATLAB will bring up a GUI window with three sinusoids (colored, dotted lines), the sum of these three sinusoids (the black, dashed line), and a target periodic signal (the black, solid line). The frequencies of the sinusoids are ω_0, $2\omega_0$, and $3\omega_0$, where ω_0 is the fundamental frequency of the target signal.

   As stated earlier, the goal of this problem is to adjust the amplitudes and phases of the three sinusoids to approximate the target signal as closely as possible. You can enter the amplitude and phase for each sinusoid in the spaces provide in the GUI window, or using the mouse, you can click-and-drag each sinusoid to change its amplitude and phase. In addition to displaying the three sinusoids, their sum, and the target signal, the GUI window also shows the mean-squared error between the sum and the target signals.

   Use `sinsum.m` to hand tune the amplitudes and phases of the three sinusoids to make the mean-squared error as small as you can.

   (Hint: You should be able achieve an MSE less than 0.24. You will receive +2 bonus points if you can achieve an MSE less than 0.231.)

   (Hint: In attempting to minimize the MSE you might try to adjust one sinusoid to minimize the MSE, then another, then another. After doing all three, go back and see if readjusting them in a "second round" has any benefits.)

   - [16(+2)] Include the resulting figure window in your report. (On Windows systems, use the "Copy to Clipboard" button to copy the figure, then you can simply paste it into a Word or similar document. There is also a "Print Figure" button for other systems if you can't get access to a PC.)

   Food for thought[14]*: Did you try the procedure suggested in the hint above, in which you tune each sinusoid one at a time and then return to each for a "second round" of tuning? If so, can you explain why the second round did or did not lead to any improvements? (Hint: Consider Fourier series property 12.)*

   Food for thought: By executing `sinsum(1)`*,* `sinsum(2)`*, and* `sinsum(3)`*, you can match different signals with sinusoids. Find MSE's that are as small as possible for each of these other signals.*

2. (Applying Fourier series synthesis) In this problem you will simply apply `fourier_synthesis` to a given set of Fourier coefficients and find the resulting continuous-time signal. Download

[13] Note that this function will *only* work under MATLAB 6 and higher. It is highly recommend that you use a Windows-based PC for this problem, since you need to copy the figure window into your report. Using the Windows clipboard simplifies this task significantly.

[14] "Food for thought" items are not required to be read or acted upon. There is no extra credit for involved. However, if you include something in your report, your GSI will read and comment on it. Alternatively, you can discuss "food for thought topics" in office hours.

the file `fourier_synthesis.m`. Use it to generate an approximation to the signal with the following Fourier coefficients:

$$\alpha_k = \begin{cases} -\left(\frac{2}{\pi k}\right)^2 & k = \pm 1, \pm 3, \pm 5, \ldots \\ 0 & k = 0, \pm 2, \pm 4, \ldots \end{cases} \tag{4.50}$$

Let $T = 0.1$ seconds, and generate 5 periods of the signal. Use $N = 20$, giving you 41 Fourier series coefficients. (Hint: First, define a frequency support vector, `kk=-20:20`. Then, generate `CC` from `kk` and set all even harmonics to zero.)

- Use `stem` to plot the magnitude of the Fourier coefficients. Use your `kk` vector as the x-axis.
- Use `plot` to plot samples of the continuous-time signal that `fourier_synthesis` returns versus time in seconds.
- What kind of signal is this?

3. (Applying Fourier series analysis) In this problem you will use the Fourier series analysis and synthesis formula to see how the accuracy of the approximate synthesis formula (4.6) depends on N.

   Download the files `lab4_data.mat` and `fourier_analysis.m`. `lab4_data.mat` contains the variables `step_signal` and `step_time`, which are the signal and support vectors for the samples of a continuous-time periodic signal with fundamental period $T_0 = 1$ second. Note that there are $N_s = 16384$ samples in one fundamental period. (`step_signal` and `step_time` include several fundamental periods, but you'll be dealing with only one period in several parts of this problem. As such, you might find it useful to create a one-period version of `step_signal`.)

   (a) (Look at the signal to be analyzed) First, let us examine `step_signal`.
   - Use `plot` to plot `step_signal` versus its support vector.
   - Compute the mean-squared value of `step_signal`.

   (b) (Perform FS analysis) Use `fourier_analysis` to perform a T_0 second Fourier series analysis over *a single period* of `step_signal` with with $N = 50$.
   - Use `subplot` and `stem` to plot the magnitude and phase of the resulting Fourier series coefficients. Make sure that your x-axis is given in frequency.

   (c) (Resynthesize FS approximations) Use `fourier_analysis` and `fourier_synthesis` to generate an approximations of `step_signal` with $N = 25$, 50, 100, and 200. (Perform T_0-second Fourier analysis and synthesis over a single period of the signal for each N. Be sure to resynthesize a single period with `Ns` $= 16384$ samples.)
   - Use `plot` and `subplot` to plot your resynthesized signals for each N in separate panels of a subplot array.
   - Calculate the mean-squared error of the resynthesis for each value of N.
   - Compute the sum of the squared magnitudes of `CC` for each value of N.
   - Find and document a relationship between the mean-squared errors and the sum of squared magnitudes of `CC` you have computed. (Hint: Consider the mean-squared value that you computed for `step_signal`. You might also want to look in the Properties of Fourier Coefficients subsection.)

(d) (Meet an MSE target) Find the smallest value of N for which the mean-squared error of the resynthesis is less than 0.5% of the mean-squared value of `step_signal`.

- Include this value in your report.

Food for thought: Try repeating Part (b) with the Fourier analysis performed over two fundamental periods of the signal, and compare to the previous answer to Part (b). Do the new Fourier coefficients turn out as expected?

4. (Using the DFT to describe a signal as a sum of discrete-time sinusoids) In this problem, you will simply apply the DFT to a particular discrete-time signal, which is also contained in `lab4_data.mat`, namely, `signal_id`. `signal_id` is considered to be a periodic discrete-time signal with fundamental period $N_0 = 128 =$ `length(signal_id)`. Take the N_0-point DFT of `signal_id`.

   - Use `stem` to plot the magnitude of the DFT versus the DFT coefficient index, k.
   - Use the DFT to describe `signal_id` as a sum of discrete-time sinusoids. That is, for each sinusoid, give the amplitude, frequency (in radians per sample), and phase.

5. (Use the DFT to remove undesired components from a signal) In this problem you will use the technique described in Section 4.2.4 to eliminate a noise signal from a desired signal. This signal, `melody`, is also contained in `lab4_data.mat`. This variable contains samples of a continuous-time signal sampled at rate $f_s = 8192$ samples/second. It contains a simple melody with one note every 1/2 second. Unfortunately, this melody is corrupted by another "instrument" playing a constant note throughout. We would like to remove this second instrument from the signal, and we will use the DFT to do so.

   It is a good idea to begin by listening to `melody` using the `soundsc` command.

   (a) (Examine DFT of first note) In order to remove the corrupting instrument, we need to determine where it lies in the frequency domain. Let's begin by looking at just the first note (i.e the first 0.5 seconds or 4096 samples). This "note" consists of the sum of two notes — one is the first note of the melody, the other is the constant note from the corrupting instrument. Each of these notes has components forming a harmonic series. The fundamental frequencies of these harmonic series are different, which is the key to our being able to remove the corrupting note. Take the DFT of the first 0.5 seconds (4096 samples) of the signal.

   - Use `stem` to plot the magnitude of the DFT for the first note.
   - Identify the frequencies contained in each of the two harmonic series present in signal. What are the fundamental frequencies?

   (b) (Examine DFT of second note) By comparing the spectra of the first two notes, we can identify the corrupting instrument. Take the DFT of the second 0.5 seconds (samples 4097 through 8192).

   - Use `stem` to plot the magnitude of the DFT for the second 0.5 seconds.
   - What are the fundamental frequencies (in Hz) of the two harmonic series in this note?
   - We know that the melody changes from the first note to the second, but the corrupting instrument does not. Thus, by comparing the harmonic series found in this and the previous part, identify which fundamental frequency belongs to the melody and which to the corrupting instrument.

(c) (Identify the DFT coefficients of the corrupting signal) In order to remove the "corrupting" instrument, we simply need to zero-out the coefficients corresponding to the harmonics of the note from the corrupting instrument. This is done directly on the DFT coefficients of each 0.5 seconds of the signal. Then, we resynthesize the signal from the modified DFT coefficients.

- Based on this, and your results from the previous parts of this problem, which DFT coefficients need to be set to zero in order to remove the corrupting instrument from this signal? (Hint: Remember the conjugate pairs.)

(d) (Complete the function that removes the corrupting instrument) Finally, we'd like to remove the corrupting instrument from our melody. Download the file `fix_melody.m`. This function contains the code that you'll use to remove the corrupting instrument from the melody signal. For each note of the melody, the function takes the DFT, zeros out the appropriate coefficients (which you must provide), and resynthesizes the signal.

- Complete the function by setting the variable `zc` equal to a vector containing the DFT coefficients that must be zeroed-out.
- Execute the function using the command

  ```
  >> result = fix_melody(melody);
  ```

  Listen to the resulting signal. Have you successfully removed the corrupting instrument?

(e) (Check your result with the spectrogram) Finally, we'd like to be able to visually check our result. Download the function `melody_check.m`. `melody_check` produces an image called a *spectrogram* that you can use to check your work. Basically, the spectrogram works by taking the DFT of many short segments of a signal and arranging them as the columns of an image. Note that the x-axis is time and the y-axis is frequency. The color of each point on the image represents the strength of the spectral component (in decibels) at that time and frequency. The dark horizontal bands show the presence of sinusoidal components in the signal at the associated times.

- Execute `melody_check` by passing it `melody`. Include the resulting figure in your report.
- Can you identify the components of the corrupting instrument on this spectrogram?
- Now, execute `melody_check` by passing it `result`. Include the resulting figure in your report.
- Compare the spectrogram of `melody` to the spectrogram of `result`. What differences do you see? Is this what you expect to see?

6. On the front page of your report, please provide an estimate of the average amount of time spent outside of lab by each member of the group.

Laboratory 5

Images, Compression, and Coding

5.1 Introduction

A common application of signals and systems is in the production, manipulation, storage and distribution of images. For example, image transmission is an important aspect of communication (especially on the internet), and we would like to be able to distribute high quality images quickly over low-bandwidth connections. To do so, images must be *encoded* into a sequence or file of bits, which can be digitally transmitted or stored. When display of the image is required, the sequence/file of bits must be *decoded* into a reproduction of the image. A block diagram of a general data compression system, with an encoder and decoder, is shown in Figure 5.1.

Systems or algorithms that do the encoding and decoding are called *source coders*, *coders*, *data compressors*, or *compressors*. They are called *source coders* because they encode the data from a *source*, e.g. a camera or scanner. They are also called data compressors, because their encoders usually produce fewer bits than were produced by the original data collector. For example, JPEG is a commonly used, standardized image compressor. You've probably downloaded many JPEG encoded images over the internet — any image with filename extension .jpg. FAX machines use a different image compression algorithm.

In this lab, we will experiment with some basic data compression techniques as applied to images. Typically, there is a tradeoff between the number of bits an encoder produces and the quality of the decoded reproduction. With more bits we can usually obtain better quality at the expense of greater storage or bandwidth requirements. When we assess how well these techniques work, we will count the number of bits their encoders produce(fewer is better), and as a measure of quality, and we will compute the mean-squared or RMS error as a measure of the quality of the decoded reproduction (low error means high quality, or equivalently, low distortion).

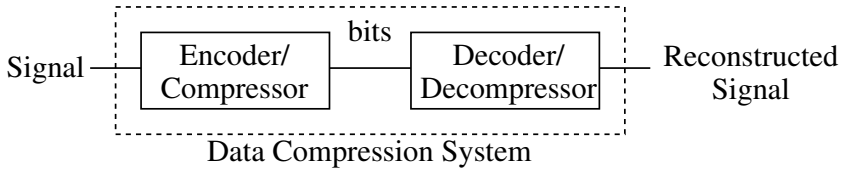


Figure 5.1: A block diagram of a general data compression system.

5.1.1 "The Question"

- How can we compress images so that they take up less storage space and/or less bandwidth?

5.2 Background

5.2.1 Images

So far, we have dealt entirely with *one-dimensional* signals. That is, these signals are indexed by only one independent variable (usually time). In this lab, we will start to consider *two-dimensional* signals. An image is an example of a two-dimensional signal. In an image, we usually index the signal based on horizontal and vertical position — two dimensions that are needed to find the "signal value" at any given point.

In this lab, we will generally restrict our attention to gray-scale images[1]. We mathematically represent such an image (in *continuous-space*) as a signal $x(t, s)$, where $0 \leq t \leq H$, $0 \leq s \leq W$. $x(t, s)$ denotes the *intensity*, *brightness*, or *value* of the image at the position with vertical coordinate t and horizontal coordinate s, and H and W are the height and width of the image, respectively. The values of $x(t, s)$ are generally nonnegative. Thus, a small value of $x(t, s)$ (close to zero) corresponds to black while larger values correspond to progressively lighter shades of gray.

In *digital image processing*, the image is assumed to be sampled at regularly spaced intervals creating a *discrete-space* image $x[m, n]$:

$$x[m, n] = x(mT_s, nT_s), \tag{5.1}$$

where T_s is the sampling interval, given in units of *distance*. Thus, in discrete-space, an image is simply an $M \times N$ array or matrix of numbers $x[m, n]$, where m and n are integers in the range $[1, M]$ and $[1, N]$, respectively. Each $x[m, n]$ is called a *pixel*. We adopt the usual convention that $x[1, 1]$ is the upper left pixel, $x[1, N]$ is the upper right, $x[M, 1]$ is the lower left, and $x[M, N]$ is the lower right.

We shall also adopt the common, but not universal, convention of digital image processing that pixel values, often called *levels*, are integers ranging from 0 to 255. The reason the pixel values are integers is that computers cannot store real-valued quantities. Instead the raw pixel values must be *quantized* to values from a finite set. The usual practice is to scale the raw image pixel values by some constant so the maximum value is close to 255 and then to round each pixel value to the nearest integer, thereby obtaining an image whose values are integers between 0 and 255. Why 0 to 255? There are two reasons. One is that these values can be conveniently represented with one byte, i.e. 8 bits.[2] Another reason is that the effects of rounding to 256 possible levels are not ordinarily observable, whereas rounding to a significantly smaller number, say 128, is sometimes noticeable.

5.2.2 Signal statistics for images

Two-dimensional signals in general, and images in particular, have the same sorts of statistics that one-dimensional images have. Generalizing from the one-dimensional case is often quite straightforward. We will also introduce two new statistics for both one- and two-dimensional signals.

[1] Color images are often represented as three *separate* signals (or *channels*), one each for red, green, and blue.

[2] To store an integer in a computer, it must be represented with a binary sequence. If binary sequences of length b are used, then 2^b levels can be represented, because there are 2^b distinct binary sequences of length b. Thus, it takes 8 bits to store the 256 levels from 0 to 255.

1. **Average Value.** The *mean* or *average value*, M, of a discrete-space image $x[m,n]$ is

$$M(x) = \frac{1}{NM}\sum_{m=1}^{M}\sum_{n=1}^{N} x[m,n] \tag{5.2}$$

2. **Mean-squared value.** The *mean-squared value* (or MSV), MS, of a discrete-space image $x[m,n]$ is

$$MS(x) = \frac{1}{NM}\sum_{m=1}^{M}\sum_{n=1}^{N} x^2[m,n] \tag{5.3}$$

3. **Root mean-squared value.** The *root mean-squared value* (or RMS value) of a discrete-space image $x[m,n]$ is

$$\begin{aligned} RMS(x) &= \sqrt{\frac{1}{NM}\sum_{m=1}^{M}\sum_{n=1}^{N} x^2[m,n]} && (5.4)\\ &= \sqrt{MS(x)} && (5.5)\end{aligned}$$

4. **Variance.** The *variance* of a discrete-space image $x[m,n]$ is

$$\begin{aligned} Var(x) &= \frac{1}{NM}\sum_{m=1}^{M}\sum_{n=1}^{N}(x[m,n]-M(x))^2 && (5.6)\\ &= MS(x-M(x)) && (5.7)\\ &= MS(x)-(M(x))^2 && (5.8)\end{aligned}$$

5. **Standard deviation.** The *standard deviation* of a discrete-space image $x[m,n]$ is

$$\begin{aligned} Std(x) &= \sqrt{\frac{1}{NM}\sum_{m=1}^{M}\sum_{n=1}^{N}(x[m,n]-M(x))^2} && (5.9)\\ &= \sqrt{Var(x)} && (5.10)\\ &= RMS(x-M(x)) && (5.11)\\ &= \sqrt{MS(x)-(M(x))^2} && (5.12)\end{aligned}$$

Notice the relationship between the variance and standard deviation, equation (5.10), and the relationship between these statistics and the MS and RMS values[3], equations (5.8) and (5.12). The variance and standard deviation measure how widely varying are the values of a signal. If they are small, it means that the signal values (and thus the signal value distribution) is tightly clustered around the mean value, while if they are large, the signal values range widely.

Recall from Laboratory 1 that we often use the MS and RMS values to measure *distortion* of a signal. We will be doing this for images in this laboratory. If $y[m,n]$ is a distorted version of $x[m,n]$, then we can measure the *mean-squared error* (MSE) and *root mean-squared error* (RMSE), using

$$\begin{aligned} MSE &= \frac{1}{NM}\sum_{n=1}^{N}\sum_{m=1}^{M}(x[m,n]-y[m,n])^2 && (5.13)\\ RMSE &= \sqrt{\text{MSE}}. && (5.14)\end{aligned}$$

[3]Equation (5.8) is not something that is immediately obvious, but it is something that can be straightforwardly derived. (Doing so is an interesting exercise.)

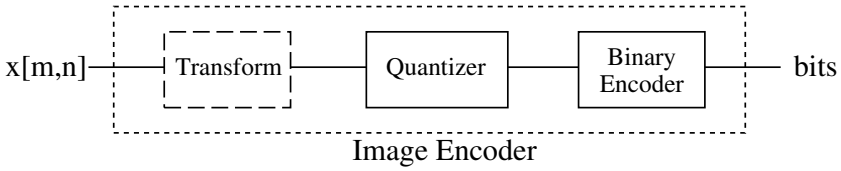


Figure 5.2: A block diagram of a general image encoder/compressor.

5.2.3 Data compression

There are two primary types of data compressors: *lossless* and *lossy*. A lossless compressor will encode and decode in such a way that the decoded reproduction is exactly the same as the original (8 bits per pixel) image. A lossy compressor will encode and decode in such a way that the decoder produces only an approximation to the original image. On the one hand, lossless is better because it is, well, lossless. This is essential when compressing computer files. UNIX *compress*, *gzip*, and the PKZip compression formats are all lossless.

On the other hand, if a small amount of distortion is permitted, lossy compressors can attain much larger amounts of compression, i.e. their encoders can produce many fewer bits. For multimedia, lossy compression is often acceptable. Examples of lossy compression schemes that you may have used include MP3 (for audio), JPEG (for photos), and MPEG (for movies). These three schemes are all examples of *transform coding* methods; we will examine a simple transform coding scheme in this laboratory. MP3 encoding is also an example of so-called *perceptual coding*. Perceptual coding methods often introduce significant amounts of distortion, but do so in a way that is nearly imperceptible.

Consider the generalized image encoder/compressor shown in Figure 5.2. It consists of three main components: the *transform*, the *quantizer* and the *binary encoder*. These are described briefly now, and in more detail in the next three subsections. The input to the encoder is a sampled image, $x[m, n]$. We generally consider that the pixels of $x[m, n]$ take on a continuum of real values.

The first component, the *transform*, is optional. When it is included, it is usually a spatial-domain to frequency-domain transformation like the Discrete Fourier Transform (DFT). Applying the transform to short segments or *blocks* of the signal tends to concentrate the energy of the signal into just a few coefficients, which permits the quantizer and binary encoder to be more effective.

The next component is *quantization*. Quantization takes the input sample/pixel and "rounds" it to one of a finite set of *levels*. It is a lossy or noninvertible operation in the sense that one cannot recover the original sample/pixel from the quantized sample/pixel. As an example, we noted previously that digital images often have pixel values ranging from 0 to 255. This is because each *raw* pixel value, as produced by some camera, has already been rounded to the nearest of a set of 256 quantization levels. Lossless image compression schemes work by operating on image that are already quantized; additional quantization is not permitted. However, in lossy compression schemes, additional quantization is performed, in order to obtain greater compression. For example, each image pixel may be quantized to the nearest of a set of only 64 levels.

The final component is *binary coding*, which assigns a sequence of bits called a *codeword*, to each level produced by the quantizer. In some systems, called *fixed-length coders*, the number of bits used to represent each pixel is known in advance (e.g. 2 bits per pixel). This is the simplest type of coder. Other systems, called *variable-length coders*, assign binary codewords of different lengths to each pixel value, usually based on the frequency of occurrence. More frequently occuring levels are assigned shorter codewords. This allows the compression system to achieve additional compression.

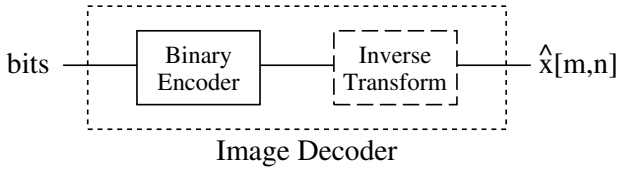


Figure 5.3: A block diagram of a general image decoder/decompressor.

Many advanced compression systems, including JPEG, use variable length coders. MP3 coding has provisions for both fixed-length and variable-length coding.

As illustrated in Figure 5.3, the decoder/decompressor corresponding to the encoder/compressor just described has two components. The input to the decoder is the bits produced by the encoder. The first component, the *binary decoder*, inverts the operation of the binary encoder, and produces the levels originally produced by the quantizer. The quantization operation produced by the encoder is not invertible, so there is not corresponding decoding step. Instead the last step is the inverse transform, which as the name suggests performs the inverse of the encoding transform. The output of the decoder, called an *encoded or decoded image or reproduction* can be displayed on a monitor or printed on paper, as desired.

5.2.4 Transformation

Efficient lossy data compressors typically perform some sort of preprocessing on the data to be compressed. One very common preprocessing step is a *transform*, and such compressors are called *transform coders*. For example, JPEG is a transform coder based on the discrete cosine transform (DCT), which is a spectral transformation similar to the DFT. The transform is typically applied to small groups of pixels called *blocks*. In this lab, we will experiment with a simple DFT-based transform coder that uses short 1×8 pixel blocks. That is, we use an N-point DFT with $N = 8$. Recall that the synthesis and analysis formulas for an 8-point DFT are given by

$$x[n] = \sum_{k=0}^{7} X[k] e^{j\frac{2\pi k}{8}n} \tag{5.15}$$

$$X[k] = \frac{1}{8} \sum_{n=0}^{7} x[n] e^{-j\frac{2\pi k}{8}n} \tag{5.16}$$

Here, $X[k]e^{j\frac{2\pi k}{8}n}$ is the "spatial" frequency component at frequency $\hat{\omega} = \frac{2\pi k}{8}$. The DFT synthesis formula shows that an image block $x[n]$ can be viewed as the sum of such components. More specifically,

$$\begin{aligned}(x[0], x[1], x[2], \ldots, x[7]) = \; & X[0]\,(1,\ 1,\ 1,\ \ldots,\ 1) \\ & + X[1]\,(\,e^{j\frac{2\pi}{N}0},\ e^{j\frac{2\pi}{N}1},\ e^{j\frac{2\pi}{N}2},\ \ldots,\ e^{j\frac{2\pi}{N}7}\,) \\ & + X[2]\,(\,e^{j\frac{2\pi 2}{N}0},\ e^{j\frac{2\pi 2}{N}1},\ e^{j\frac{2\pi 2}{N}2},\ \ldots,\ e^{j\frac{2\pi 2}{N}7}\,) \\ & + \ldots \\ & + X[7]\,(\,e^{j\frac{2\pi 2}{N}0},\ e^{j\frac{2\pi 2}{N}1},\ e^{j\frac{2\pi 2}{N}2},\ \ldots,\ e^{j\frac{2\pi 2}{N}7}\,)\end{aligned} \tag{5.17}$$

Note that we are NOT presuming that these blocks are in any way periodic.

Why do we perform a spectral transformation before compressing a signal? As suggested in Section 5.2.3, the transformation serves to shift around the energy in a given block so that it is easier to compress. Consider, for instance, a single block of an image given by

$$x[n] = (\ 165,\ 168,\ 167,\ 166,\ 167,\ 165,\ 168,\ 166\ )$$

This block is roughly constant, so we expect its 8-point DFT to have a large $X[0]$ (i.e., DC) component. Since there is little other variation, though, the rest of the DFT coefficients will be relatively small. By only storing the $X[0]$ coefficient, for instance, and throwing away the rest, we only need $1/8^{\text{th}}$ as much storage as if we had stored all of the coefficients for this block; further, we have introduced only a small amount of distortion (as measured by the mean-squared error). While we won't go so far as to throw away the rest, this example suggests the basic idea for how to use the transform to compress a signal. If some transformed coefficients tend to be smaller than others, we can store them more efficiently.

An 8-point DFT produces 8 complex numbers, $X[0], X[1], \ldots, X[7]$. This actually translates into 16 real numbers (the 8 real and 8 imaginary parts) that we need to consider storing. Thus taking the transfrom would at first seem to be a bad idea, because we now need twice as much storage to represent a block! However, there are symmetry properties that we can exploit so that we only need to store 8 of these numbers. Since the input signal is real, recall that the 8-point DFT, $X[k]$ has the conjugate symmetry property:

$$X[8-k] = X^*[k] \tag{5.18}$$

This means that knowing the real and imaginary parts of $X[1]$, for instance, completely determines the real and imaginary parts of $X[N-1]$. Thus, though we need to store the real and imaginary parts of $X[0], X[1], X[2], X[3], X[4]$, we do *not* need to store $X[5], X[6], X[7]$, because these values can be recovered from the previous four. Further, the coefficients $X[0]$ and $X[4]$ are purely real (that is, $X[0] = X^*[0]$ and $X[4] = X[8-4] = X^*[4]$). Thus, $X[0]$ and $X[4]$ each require the storage of a single real number. From this argument, we can see that the 8-point DFT produces a total of only eight real numbers that must be stored.

In our implementation of the transform encoder, the eight numbers $c[0], \ldots, c[7]$ that we choose to represent the 8-point DFT $X[0], \ldots, X[7]$, of an image block are

$$\begin{aligned}
c[0] &= X[0] \\
c[1] &= \sqrt{2}\ \text{Re}\{X[1]\} \\
c[2] &= \sqrt{2}\ \text{Re}\{X[2]\} \\
c[3] &= \sqrt{2}\ \text{Re}\{X[3]\} \\
c[4] &= X[4] \\
c[5] &= \sqrt{2}\ \text{Im}\{X[1]\} \\
c[6] &= \sqrt{2}\ \text{Im}\{X[2]\} \\
c[7] &= \sqrt{2}\ \text{Im}\{X[3]\}
\end{aligned}$$

The $\sqrt{2}$ factors have been included where one coefficient is, in effect, standing in for two. It can be shown that with these factors

$$\sum_{n=0}^{7} x^2[1,n] = 8 \sum_{k=0}^{7} c^2[k], \tag{5.19}$$

which is an often useful fact. This is derived using Parseval's relation, as given in Lab 4[4].

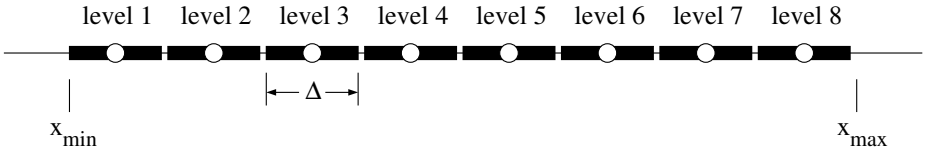
[4]For this derivation, see the document titled "Notes: The Distortion of Transform Coding" by D.L. Neuhoff.

5.2.5 Quantization

Quantization is the most elementary form of lossy data compression, while also forming a fundamental part of more advanced lossy compression schemes such as transform coding. We may quantize an image directly, or we may quantize the results of a transformation as described in Section 5.2.4. When a number x is *quantized to L levels*, we mean that its value is replaced by (or quantized to) the nearest member of a set of L *quantization levels*. Here, we consider *uniform quantization*[5]. For the uniform quantization used here:

- We define a *quantizer range* defined by values x_{min} and x_{max}
- We divide this range into L equally sized segments, each with size $\Delta = \frac{x_{max}-x_{min}}{L}$.
- We place the quantization level for a given segment in the middle of that segment.

The quantizer is illustrated with the figure shown below, which shows $L = 8$ segments of width $\Delta = (x_{max} - x_{min})/8$ as thick lines and the corresponding levels within each segment as circles.



Given a pixel $x[m, n]$, the quantizer operates by outputing the nearest level. Equivalently, if $x[m, n]$ lies in the i^{th} segment, then quantizer outputs the i^{th} level. If x is larger than x_{max}, then x is quantized to the largest level, namely, $x_{max} - \Delta/2$. Similarly, if x is smaller than x_{min}, then x is quantized to the smallest level, namely, $x_{min} + \Delta/2$.

One can see that if x is within the quantizer range, then its quantized value will differ from x by at most $\Delta/2$, so that the quantizer introduces only a small error. On the other hand, when x is outside the range, the quantizer can introduce a large error. Thus, when designing a quantizer it is important to choose the quantizer range so that it includes most values of x. Making the range large will do this. However, we don't want to make the range too large. Larger ranges mean that $\Delta = (x_{max} - x_{min})/L$ is larger, which in turn increases the maximum possible error introduced when x lies within the range of the quantizer.

5.2.6 Binary coding

The output of a data compression encoder must always be bits, not quantized samples or pixels. Thus, the quantizer is always followed by a *binary encoder*, as illustrated in Figure 5.4. A compressor that consists simply of a quantizer followed by a binary encoder will be called a *direct quantizer*, in contrast to a transform coder or some other coder that involves a preprocessing step.

A binary encoder operates by assigning a distinct sequence of bits, called a *codeword* to each level of the quantizer. For example, an assignment of codewords to levels is shown below

Such binary codewords are the output of the encoder when quantizing the data. The decoder will, eventually, receive a binary codeword and output the corresponding quantization level as the

[5]There are sometimes advantages to using quantizers with unequal level spacings, but we will not deal with such quantizers in this lab. Uniform quantizers are sometimes called *uniform scalar quantizers* to distinguish them from more sophisticated quantizers that do not operate independently on successive data samples.

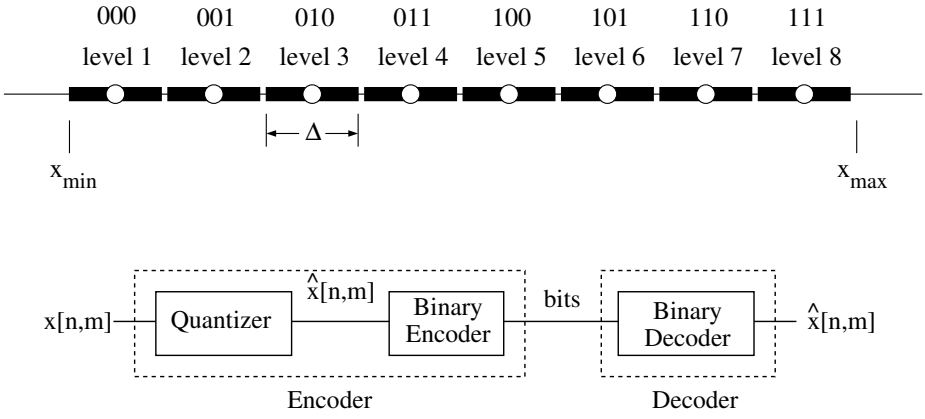


Figure 5.4: Block diagram of a direct quantizer

reproduction of the original piece of data. For instance, if the image pixel $x[m, n]$ lies in the third segment of the quantizer, the binary encoder will produce 010, which when received, the decoder will produce level 3 as the reproduction of $x[m, n]$.

If, as often happens, the number of levels is a power of two, i.e. $M = 2^b$ where b is an integer, then the simplest approach is to make each codeword have b bits. It does not matter which b-bit sequence is assigned to which level, but the usual scheme, as illustrated above, is to assign the binary sequence representing 0 to the smallest level, the binary sequence representing 1 to the next largest level, and so on. With this type of binary coding, the encoder is fixed-length (or *fixed-rate*) in the sense described earlier. Often, a better scheme is to use shorter codewords for the quantization levels that occur more frequently, and longer ones for those that are used less frequently. Such variable-length codes are used in JPEG and other high efficiency schemes.

5.2.7 Performance

There are two ways that we measure the performance of a compression system. First, we want to know how many bits are required to store an image. The total number of bits produced by the encoder is equal to the number of blocks multiplied by the number of bits required to encode one block. More commonly, we report the number of bits required to store a single pixel. This is called the *coding rate*, R. The coding rate is equal to the number of bits required to code a single block divided by the number of pixels in a block. Naturally, we prefer a lower coding rate.

The second performance measure is the amount of distortion introduced by the coder. Generally, we measure this distortion by computing the mean-squared (MSE) or RMS error (RMSE). We also prefer to have low distortion, and equivalently low error.

Unfortunately, we generally have to trade off between these two performance measures. That is, we can produce a highly compressed (with a low coding rate) image, but this generally introduces a large RMS error. Alternatively, we can have a very high-quality representation of an image (with low distortion), but such a representation requires many bits to encode. Figure 5.5 shows an illustration of the tradeoff between the two performance measures. In the laboratory assignment, you will produce a plot similar to this for compression using uniform quantization.

Performance of direct quantizers

Let us now analyze the performance of a direct quantizer, where the quantizer is uniform with $L = 2^b$ levels and range $[x_{min}, x_{max}]$.

Since the binary encoder for such a system assigns b bits to each level, the coding rate is

$$R = b \text{ bits/pixel (bpp)} \tag{5.20}$$

Elementary theory predicts that when the quantizer range includes most values of the image $x[m, n]$ and when Δ is much smaller than the standard deviation of the image, then the MSE induced by quantizing with level spacing Δ can be approximated as follows[6]/

$$\begin{aligned} MSE &\approx \frac{1}{12}\Delta^2 && (5.21)\\ &= \frac{1}{12}\left(\frac{x_{max} - x_{min}}{L}\right)^2 && (5.22)\\ &= \frac{1}{12}(x_{max} - x_{min})^2 2^{-2R}\,, && (5.23) \end{aligned}$$

This shows that if we were to shrink Δ by a factor of 2, as would happen if L were doubled and the range were held constant, then the MSE would decrease by a factor of four. Equivalently, the last equation shows that this factor of four reduction comes by increasing the coding rate by one bit per pixel.

When a quantizer is applied to data whose signal value distribution is fairly constant over a given range, then it is usually good practice to choose the quantizer range to match the data range. This is generally the case when directly quantizing images, so we will generally choose $x_{min} = 0$ and $x_{max} = 255$.

On the other hand, when quantizing data whose signal value distribution is quite uneven, then it may be best to choose the quantizer range to be a subset of the data range. For example, in transform coding, it often happens most of the data to be quantized is near zero but there are a few very, very large values. In such cases, experience has shown that to design a quantizer with small MSE, one should normally choose the width of the range to be proportional to the standard deviation of the data being quantized, i.e. $(x_{max} - x_{min}) = c \times Std(x) = c\sqrt{Var(x)}$. The constant of proportionality c is usually between 2 and 6. Smaller values of c work well for smaller values of L, and larger values

[6] See the document "Note: The $\Delta^2/12$ Formula" by D.L. Neuhoff.

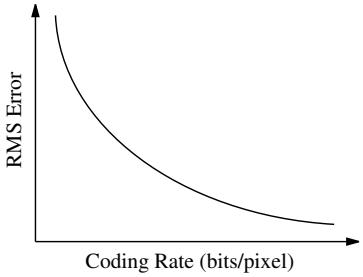


Figure 5.5: There is an inherent tradeoff between coding rate and distortion.

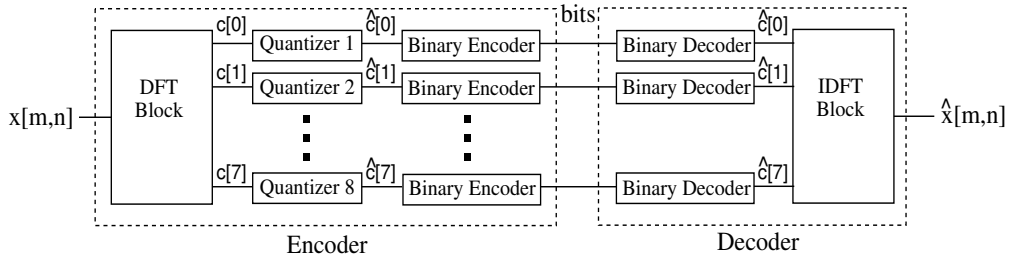


Figure 5.6: A block diagram of a transform coder. The encoder divides the incoming image, $x[m,n]$ into 1×8 blocks and transforms each block into a sequence of 8 coefficients $c[0], \ldots, c[7]$. These coefficients are then quantized to yield $\widehat{c}[0], \ldots, \widehat{c}[7]$, and encoded into a binary representation. The decoder creates a reconstruction of the image, $\widehat{x}[m,n]$ by decoding the binary codewords and inverting the transformation.

of c work well for large values of L. Using this relation in (5.21), we find

$$\text{MSE} \approx \frac{1}{12}\Delta^2 = \frac{1}{12}\left(\frac{x_{max} - x_{min}}{L}\right)^2 \tag{5.24}$$

$$\approx \frac{c^2}{12}\frac{Var(x)}{L^2}. \tag{5.25}$$

$$\approx \frac{c^2}{12} Var(x) 2^{-2R}. \tag{5.26}$$

This shows that quantizer MSE is proportional to the variance of the data and inversely proportional to L^2.

5.2.8 Designing a transform coder

In the previous sections, we have described the three main components of transform type data compression system. In particular, in Section 5.2.4, we described a transform that uses an 8-point DFT of 1×8 blocks of image pixels. A block diagram of the system based on this transform can be found in Figure 5.6.

To make this transform coder work well, though, the quantizers must be individually designed for each of the eight types of (independent) coefficients. Indeed, if we quantize all eight types of coefficients with the same number of levels, then the transform coder will not work substantially better than direct quantization (quantization without preprocessing). Thus, for each of the eight types of coefficients, we must carefully choose the number of quantization levels, L, and the quantizer range limits, x_{min} and x_{max}.

Let L_k be the number of levels for each coefficient $c[k]$. Further, let each L_k be a power of two such that $L_k = 2^{b_k}$, where b_k is the number of bits that we allocate to the transformed coefficient $c[k]$.

It should be clear that choosing large L_k's will permit the transform coder to encode with less distortion. However, the total number of bits produced by the encoder is the number of blocks, $\frac{N^2}{8}$, times the number of bits to encode one block, $\sum_{k=0}^{7} b_k$. Thus, higher L_k's require more bits to store the signal, and thus a higher coding rate. For this transform coder, we can calculate the coding rate,

R, as

$$R = \frac{1}{8}\sum_{k=0}^{7} b_k \text{ bpp} . \tag{5.27}$$

In many situations, we are given a desired coding rate R, e.g. $R = 2$ bpp. In this case, the question becomes how we should divide these bits among the eight types of coefficients, i.e. how to choose the b_k's, so they average to the desired coding rate R, yet cause the distortion in the reproduction produced by the transform code to be as small as possible.

Using (5.19), it can be shown that[7].

$$\text{MSE} = \sum_{k=0}^{7} \text{MSE[k]} , \tag{5.28}$$

where MSE$[k]$ is the MSE of the quantizer for $c[k]$. In other words, the MSE of the transform coder is approximately the sum of the MSE's of the quantizers for the different coefficients.

Let us first consider a transform coder where each type of coefficient is quantized with the same number of bits/pixel, i.e. $b_0 = b_1 = \ldots = b_7$. We assert without proof that such a transform coder has roughly the same MSE as that of direct quantization with the same number of bits/pixel. Now, we will now argue that changing the $b[k]$'s so that some are larger than others will make the transform coder work better than direct quantization.

From (5.26) we have that

$$\text{MSE[k]} \approx \frac{1}{12}\, c^2\, Var(c[k]) 2^{-2b_k} , \tag{5.29}$$

where $Var(c[k])$ denotes the variance of the $c[k]$ values. One can see from the above that the coefficients with larger variance will be quantized with larger mean-squared error. In particular the DC coefficients $C[0]$ usually have the largest variance; so they will have the largest MSE. On the other hand, the $c[3]$'s and $c[7]$'s usually have the smallest variance and distortion.

Now suppose we increase b_0 by one and decrease b_7 by one. From (5.27) we see that this will have no net effect on the number of bits produced by the coder. However, from (5.29) we see that this decreases the (large) MSE of the DC coefficients $c[0]$ by a factor of 4, and increases the (small) MSE of the $c[7]$ coefficients by a factor of 4. Is it beneficial to decrease one MSE by 4, when another one increases by 4? We can see from (5.28) that indeed it is beneficial. Decreasing a larger MSE by the factor 4 decreases the average in (5.28) more than increasing a small MSE by the factor of 4 increases the average.[8] Thus, what we want to do is shift bits towards the coefficients with larger variances. This will make MSE smaller than if all coefficients were quantized with the same number of bits and, therefore, smaller than the distortion of direct quantization.

More generally, in a well designed transform code, all of the MSE$[k]$'s will be approximately the same. If they were quite different, we could move a bit from a coefficient with small MSE to one with large MSE and achieve a net decrease in overall MSE. In this light, we can see that the role of the transform is to make the variances of the coefficients as different as possible. Some should be large, and others should be small.

5.3 Some MATLAB commands for this lab

- **Making a matrix into a vector.** Especially when working with images, it is often useful to be able to convert a matrix into a vector containing the same elements. In MATLAB, we can

[7] See the document titled "Notes: The Distortion of Transform Coding" by D.L. Neuhoff.

[8] For example, $24 + 4$ is larger than $24/4 + 4 \times 4$.

do this for a matrix `x` in the following manner:

```
>> y = x(:);
```

After this operation, `y` contains a "vectorized" version of `x`. Specifically, if `x` is an $M \times N$ matrix, `y` is a vector whose first M elements are the first column of `x`, whose second M elements are the second column of `x`, and so on. This is especially useful for calculating many of the signal statistics presented in this laboratory.

- **Calculating signal statistics on images.** To compute these statistics on images, we first need "vectorize" the image.

  1. Average value, $M(x)$:

     ```
     >> M = mean(x(:))
     ```

  2. Mean-squared value, $MS(x)$:

     ```
     >> MS = mean(x(:).^ 2)
     ```

  3. Root-mean squared value, $RMS(x)$:

     ```
     >> RMS = sqrt(mean(x(:).^ 2))
     ```

  4. Variance, $Var(x)$.

     ```
     >> variance = var(x(:));
     ```

  5. Standard deviation, $Std(x)$.

     ```
     >> std_dev = std(x(:));
     ```

  6. Signal value distribution. To compute a histogram with 256 bins centered at integers from 0 to 255, use the command

     ```
     >> hist(x(:),0:255);
     ```

- **Loading images.** Images are generally stored in some sort of standard file format, like TIFF or JPEG. To load such an image file into MATLAB, we use the command `imread`. Unfortunately, `imread` generally returns images as arrays of integers. This is unfortunate because MATLAB puts some heavy restrictions on the use of integers. In particular, to prevent integer overflow you cannot perform arithmetic on integers. Thus, we need to convert our loaded images into double precision arrays using the command `double`. To load and convert an image in the file `my_img.tif`, for instance, use the command

  ```
  >> x = double(imread('my_img.tif'));
  ```

  Note that the `imread` command will load many standard image file formats, including JPEG, PNG, BMP, TIFF, PCX, and a host of others.

- **Displaying images.** To display an image in MATLAB, there are actually a number of commands that must be used simultaneously. To display the image itself, we use `imagesc` command. To tell MATLAB to display the image as a gray-scale image, we use command `colormap(gray)`. To set the axes so that the aspect ratio is correct, use the command `axis image`. Finally, to add a "color bar" that relates image values to colors, use the `colorbar` command. *Every image that you produce for this course must have a color bar; you will lose points for* every *image you display without a color bar.* To do all of these things at once to display an image `x`, use the following code:


```
>> imagesc(your_img); colormap(gray); axis image; colorbar
```

You will be using this sequence of commands often, so you might wish to write a short function that executes all of these commands simultaneously.

- **Quantizing an image:** The function `quantize_fcn.m`, which we provide to you for this lab, implements a uniform quantizer for images and transform coefficients. It takes a signal, the desired number of quantization levels (L), and the two numbers that define the quantization range, x_{min} and x_{max}. For instance, to quantize an image, `img`, to 64 levels, use the command

  ```
  >> [q_img, delta] = quantize_fcn(img,64,0,255);
  ```

  `q_img` contains the quantized image, while `delta` contains the Δ value used for quantization. Here, note that $x_{min} = 0$ and $x_{max} = 255$. This separately quantizes each pixel of `img` to one of 64 levels, in accordance with the procedure described in the background section.

- **Using the DFT Coder:** The DFT-based transform coder that we have described in this laboratory is provided as three separate functions.

  - `dft_block.m` breaks the image into 1×8 blocks and computes the DFT of each block. If the image is $N \times N$, this function produces a series of eight *band images*. For $k = 1, \ldots, 8$, the k^{th} band image contains the $c[k-1]$ coefficients for each block. For example, the $k = 1$ band image, contains the $c[0]$, or DC, coefficients from each block. Each band image has size $M \times N/8$.

    The eight band images are returned as a three-dimensional array. To produce the band images for an image, `img` and then access the third band image, for instance, we would use the commands

    ```
    >> A = dft_block(img);
    >> A(:,:,3);
    ```

    Note that except for the first one, each band image contains both positive and negative values. However, we can still display them using `imagesc`.

  - `inverse_dft_block.m` reconstructs the image from the matrix of band images returned by `dft_block.m`.

  - `dft_coder.m` puts both of these blocks together by calling `dft_block`, quantizing the coefficient matrix, and reconstructing the image with `inverse_dft_block`.

    `dft_coder` takes several input parameters, all of which are optional except the first one. The first parameter is the image to encode. The second is a vector of bit allocations, b_k. For instance, if we call `dft_coder` like this

    ```
    >> coded = dft_coder(img,[8 6 6 6 6 4 4 4]);
    ```

    we quantize our $c[0]$ (DC) coefficients using 8 bits, the next four (real) coefficients with 6 bits each, and the last three (imaginary) coefficients using 4 bits each[9].

[9] Though more advanced coders may allow the allocation of fractions of bits, for this coder you must allocate a whole number of bits to each coefficient. You can, however, assign no bits to a coefficient. In this case, that coefficient is simply set to a constant value.

Note that the number of bits required to encode a single pixel is equal to the average value of all of the b_k's. Thus, the example above uses 5.5 bits per pixel.

When run, `dft_coder` returns the decoded image and also displays a table of useful statistics corresponding to each coefficient $c[k]$. To see this table, make sure that you put a semicolon at the end of your call to `dft_coder`.

5.4 Demonstrations in the Lab Section

1. Images are signals too.
2. Signal compression
3. The "Almost JPEG" DFT Coder
4. Designing the coder

5.5 Laboratory assignment

1. (Images in MATLAB) In this problem, you'll familiarize yourself with the image capabilities of MATLAB along with one particular image, the "cameraman."

   (a) (Display an image) Load the image "cameraman.tif". (If your computer does not have the Image Processing Toolbox, you'll need to download the file from the web page).
   - Display the image and include the resulting figure in your report.
   - Calculate the size of the image (the number of rows and columns) and the total number of pixels in the image.
   - Find the minimum and maximum pixel values, x_{min} and x_{max} in the image.

   (b) (Produce and interpret a histogram) Estimate the signal value distribution of this image by generating a histogram with 256 bins centered at integers from 0 to 255.
   - Include the resulting plot in your report.
   - From this histogram, what signal values occur the most often in this image?
   - In words, describe which part(s) of the image corresponds to these signal values.

   (c) (Examining signal values) It is useful to be able to think of images in terms of the signal values that make them up. Download the M-file `display_square.m`, which will help this process. Use this function to display the pixel values in several rectangular segments of the "cameraman" image. Find, approximately, the smallest rectangle of pixels that includes the black tip of the camera lens.
   - Include in your report a plot from `display_square.m` showing the pixel values of the rectangle you found.
   - From this display, what are the row and column indices of this rectangle?
   - From this display, what are minimum and maximum values within this rectangle?

   (d) (Signal representations) We know that this image takes on only integer values over a finite range, but there are still a few different ways we can represent the image. In the original file, for instance, each pixel is represented using 8 bits. In MATLAB, though, we convert the image into 64-bit double precision values.

- How many bits are required to describe the entire image at 8 bits per pixel?
- How many bits are required to describe the entire image at 64 bits per pixel?
- How many possible pixel values can a 64-bit number represent?

2. (Direct quantization) In this problem, we will experiment with direct quantization as an image compression mechanism. Download the function `quantize_fcn.m`.

   (a) (Quantize an image) Use `quantize_fcn` to quantize the "cameraman" image using 64 levels, 16 levels, and 4 levels. Assume $x_{min} = 0$ and $x_{max} = 255$.

   - Display and include in your report the three resulting quantized images along with the original using `subplot`. Again, make sure that you indicate which image is which.
   - Describe the effects of the quantization in these plots.

   (b) (Plot quantization functions) Use MATLAB to make a plot of the function being implemented by `quantize_fcn.m`. For example, for the 64 level quantizer, run `quantize_fcn(x,64,0,255)` for x ranging from 0 to 255, and plot the resulting values versus x.

   - Plot the quantization function for the 16 level quantizer.
   - Also, plot the histogram of image quantized with 16 levels, using 256 bins centered at integers from 0 and 255.

   (c) (Quantization as compression) For the 4, 16, and 64 level quantizers,

   - How many bits are needed to represent each of these quantized images?
   - How many bits are needed to represent each pixel in one of these images?

   (d) (Measuring quantization error) Find the "error image" corresponding to each of these quantized images.

   - Using `subplot`, display and include in your report the three error images in the same plot.
   - Can you see aspects of the original images in these plots?
   - Calculate the RMS error for each quantization of the image.

   (e) (Evaluating RMS error predictions) Now, we want to compare the actual RMS error for "cameraman" versus the predicted RMS error (based on the derivation in Section 5.2.7) for quantizers with 2, 4, 8, 16, 32, 64, and 128 levels.

   - Calculate the actual RMS error for each of these quantizers.
   - Calculate the predicted RMS errors for these quantizers.
   - Plot both the actual and predicted RMS error values versus the required number of bits per pixel.
   - For what number of bits per pixel is this prediction most accurate?

3. (Compression using a transform coder) In this problem, you will experiment with the DFT-based transform coder that is described in the background section.

   (a) (Create and examine band images) Download the M-file `dft_block.m`. Use it to generate the matrix of band images for the "cameraman" image.

   - Use `subplot` to simultaneously display all eight band images. Use `axis square` rather than `axis image` when you display these band images.

- Discuss the appearances of the various band images. For example, can you see any features of the original cameraman image in any or all of them?

(b) (Reconstruct a coded image) Download the M-file `inverse_dft_block.m`. Use this function to reconstruct the original image from the set of band images produced by `dft_block`.

- Compute the RMS error between the original and the transformed/inverse transformed image. (It should be negligibly small.)

(c) (Designing coders for image compression) Download the M-file `dft_coder.m`. Our goal in using `dft_coder` is to find appropriate parameters for the eight quantizers when compressing the "camerman" image. Through intelligent design, we hope to achieve lower RMS error than with direct quantization of the image using the same number of bits. We do this by allocating bits to each of our eight quantizers independently.

i. (Design a 4 bpp coder) Find a 4 bits per pixel design with as small an RMS error as you can. You should be able to get an RMS error less than 4. (Hint: As a general rule of thumb from Section 5.2.8, bigger coefficients should get more bits.)
   - What bit allocation did you use, and what was the resulting RMS error?
   - Display the reconstructed image and the error image on the same figure using `subplot`.
   - Compare your RMS error to the RMS error of 4 bits per pixel uniform quantization that you performed in problem 2e.
   - Compare the qualitative appearance of the reconstruction produced by the transform coder to that produced by the direct quantizer.

ii. (Design a 3 bpp coder) Find a 3 bits per pixel design with as small an RMS error as you can. You should be able to get an RMS error less than 6.4.
   - What bit allocation did you use, and what was the resulting RMS error?
   - Display the reconstructed image and the error image on the same figure using `subplot`.
   - Compare your RMS error to the RMS error of the 3 bits per pixel uniform quantization that you performed in problem 2e.
   - Compare the qualitative appearance of the reconstruction produced by the transform coder to that produced by the direct quantizer. (Note: You have not yet displayed the 3 bpp image, so you will need to generate it for comparison.)

iii. (Design a 2 bpp coder) Find a 2 bits per pixel design with as small an RMS error as you can. You should be able to get an RMS error less than 10.8.
   - What bit allocation did you use, and what was the resulting RMS error?
   - Display the reconstructed image and the error image on the same figure using `subplot`.
   - Compare your RMS error to the RMS error of a 2 bits per pixel uniform quantization that you performed in problem 2e.
   - Compare the qualitative appearance of the reconstruction produced by the transform coder to that produced by the direct quantizer.

iv. (Comment on coder design) Given your experimentation with this transform coder,
   - Comment on the relative performances of direct quantization and transform coding as the number of bits/pixel changes.

Food for Thought: In this lab, we've used a 1-dimensional transform for our coder. We can achieve significantly better compression if we use a 2-dimensional transform. MATLAB *implements a two-dimensional DFT with the command* `fft2`*. As a challenging project, consider modifying the transform coder provided here to work on* 4×4 *or* 8×8 *blocks of an image. How much compression can you achieve with this modified coder?*

4. On the front page of your report, please provide an estimate of the average amount of time spent outside of lab by each member of the group.

Postscript: JPEG Compression

In this lab, we've presented a transform coder here which uses some of the same basic ideas as JPEG compression. However, JPEG achieves much higher compression rates that what we have seen, and with much less distortion. How is this achieved? There are several modifications used in JPEG coding.

1. JPEG uses a *two-dimensional transform*. This allows much greater compaction of the data into a few transform coefficients.

2. JPEG uses a transform called the *discrete cosine transform*, which purely real, rather than the DFT. This removes some of the redundancies in our coding method.

3. JPEG uses a technique called *run-length encoding*. This allows a coder to store a "run" of similar values by indicating the value and the number of repetitions.

4. JPEG uses a variable-length coding scheme (often Huffman coding, which you may study in an intermediate programming course on data structures and algorithms) to produce a bit stream for the final coded representation.

All of these improvements allow images to be significantly compressed with relatively small distortion. For more information about JPEG coding, you might wish to look at the JPEG Tutorial:

`http://www.ece.purdue.edu/~ace/jpeg-tut/jpegtut1.html`

Laboratory 6

FIR Filtering and Image Processing

6.1 Introduction

Digital filters are one of the most important tools that signal processors have to modify and improve signals. Part of their importance comes from their simplicity. In the days when analog signal processing was the norm, almost all filtering was accomplished with RLC circuits. Now, a great deal of filtering is accomplished digitally with simple (and extremely fast) routines that can run on special digital signal processing hardware or on general purpose processors.

So *why* do we filter signals? There are many reasons. One of the biggest is *noise reduction* (which we have called *signal recovery*). If our signal has undesirable frequency components, e.g. it contains noise in a frequency range where there is little or no desired signal, then we can use filters to reduce the relative amplitude of the signal at such frequencies. Such filters are often called *frequency blocking filters*, because they block signal components at certain frequencies. For example, *lowpass filters* block high frequency signal components, *highpass filters* block low frequency signal components, and *bandpass filters* block all frequencies except those in some particular range (or band) of frequencies.

There are a wide range of uses for filtering in image processing. For example, they can be used to improve the appearance of an image. For instance, if the image has granular noise, we might want to *smooth* or *blur* the image to remove such. Typically such noise has components at all frequencies, whereas the desired image has components at low and middle frequencies. The smoothing acts as a lowpass filter to reduce the high frequency components, which come, predominantly, from the noise. Alternatively, we might want to *sharpen* the image to make its edges stand out more. This requires a kind of highpass filter.

In this lab, we will experiment with a class of filters called FIR (*finite impulse response*) filters. FIR filters are simple to implement and work with. In fact, an FIR filtering operation is almost identical to the operation of *running correlation* which you have worked with in Laboratory 2. In particular, we will examine the use of FIR filters for image processing, including both smoothing and sharpening. We will also examine their use on simple one-dimensional signals.

6.1.1 "The Question"

- How do we implement FIR filters in MATLAB?
- How can we improve the appearance of an image? Specifically, how can we remove noise or

"sharpen" an image?

6.2 Background

6.2.1 Implementing FIR Filters

FIR filters are systems that we apply to signals. An FIR filter takes an input signal $x[n]$, modifies it by the application of a mathematical rule, and produces an output signal $y[n]$. This rule is generally called a *difference equation*, and it tells us how to compute each sample of the output signal $y[n]$ as a weighted sum of samples of the input signal $x[n]$. A common form of the difference equation is given as

$$y[n] = \sum_{k=0}^{M} b_k x[n-k] \tag{6.1}$$

$$= b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] + \ldots + b_M x[n-M] \tag{6.2}$$

The b_k's are called the ***FIR filter coefficients***, and M is the *order* of the FIR filter. The set of FIR filter coefficients completely specifies an FIR filter. Different choices of the order and the coefficients leads to different kinds of filters, e.g. to lowpass, highpass and bandpass filters.

Equation (6.1) defines the class of *causal* FIR filters. A more general form is given by

$$y[n] = \sum_{k=-M_1}^{M_2} b_k x[n-k] \tag{6.3}$$

$$= b_{-M_1} x[n+M_1] + \ldots + b_{-1} x[n+1] + b_0 x[n] + b_1 x[n-1] + \ldots + b_{M_2} x[n-M_2]\,, \tag{6.4}$$

where M_1 and M_2 are nonnegative integers. Here, the order of the filter is $M_1 + M_2$. When $M_1 > 0$, the FIR filter is *non-causal*. To calculate the "present" value of $y[n_0]$, a causal FIR filter only requires "present" ($n = n_0$) and "past" ($n < n_0$) values of $x[n]$. Non-causal filters, on the other hand, require "future" ($n > n_0$) values of $x[n]$. Thus, a filter with difference equation given by $y[n] = x[n] + x[n-1]$ is causal, but a filter with difference equation given by $y[n] = x[n] + x[n+1]$ is non-causal. The distinction between causal and non-causal filters is necessary if we wish to implement one of these filters in real-time. Causal filters can be implemented in real-time, but to implement non-causal filters we generally need all of the data for a signal before we can filter it.

Compare equation (6.3) with the equation for performing running correlation between a signal $b[n]$ and $x[n]$:

$$y[n] = C(b[k], x[k-n]) = \sum_{k=-\infty}^{\infty} b[k] x[k-n]. \tag{6.5}$$

Recall that we thought of running correlation as a procedure where we "slid" one signal across the other, calculating the in-place correlation at each step. If we consider that the b_k's of an FIR filter form a signal, then the application of an FIR filter uses the same procedure with two minor differences. First, when we apply an FIR filter, we are only "correlating" over a finite range; however, we typically assume $b_k = 0$ for k outside the range $[M_1, M_2]$. Thus, we can change the limits of summation to range over $(-\infty, \infty)$ without changing the result. Second, when applying a filter,

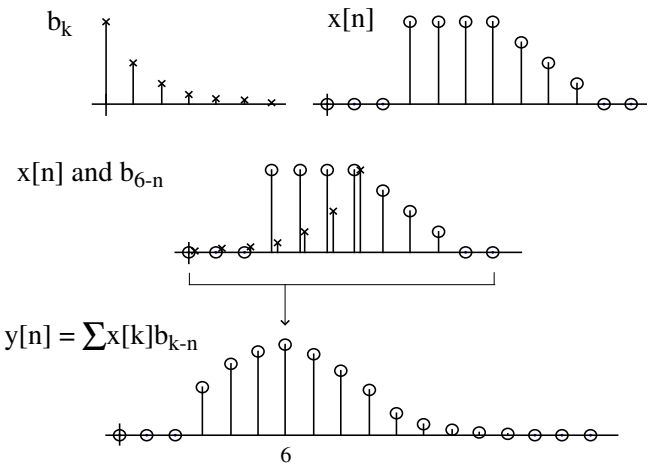


Figure 6.1: A graphical illustration of filtering. The filter coefficients, b_k and signal to be filtered, $x[n]$, are shown on the top axis. The middle axis shows $x[n]$ and a time-reversed and shifted version of b_k. We multiply these two signals and sum the result to yield a single sample of the output, $y[n]$, which is shown on the bottom axis. For example, to compute the $y[6]$ sample, we multiply the samples of $x[n]$ by b_{6-n} and sum the result.

the signal $x[n]$ is time-reversed with respect to the b_k coefficients[1]. This is not the case for running correlation.

From the definition alone, it is not easy to see how a filter "works." With the connection to correlation, though, we can suggest an intuitive graphical understanding of this process which is shown in Figure 6.1. To calculate a single sample of $y[n]$, we time-reverse the signal formed by the b_k coefficients (by flipping it across the $n = 0$ axis). Then, we shift this time-reversed signal by n samples and perform in-place correlation. The result is the n^{th} sample of $y[n]$. To build up the entire signal $y[n]$, we do this repeatedly, "sliding" one signal across the other and calculating in-place correlations at each point.

You may find it useful to go back to Lab 2 and review the algorithm for in-place correlation. In that description of the algorithm, we used $x[n]$ where here we wish to use the signal formed by the b_k's. We can use this algorithm when implementing FIR filters, as well. Note, however, that we want to time-reverse the b_k coefficients when we multiply them by the incoming signal samples. That is, we always want to multiply the b_{-M_1} coefficient by the newest sample in the buffer.

6.2.2 Edge effects and delay

Suppose that we consider filtering a signal, $x[n]$, with a causal filter whose difference equation is given by

$$y[n] = \frac{1}{5}x[n] + \frac{1}{5}x[n-1] + \frac{1}{5}x[n-2] + \frac{1}{5}x[n-3] + \frac{1}{5}x[n-4]. \tag{6.6}$$

[1]That is, $x[n-k]$ is a time-reversed version of $x[k-n]$, just as $s[-n]$ is a time-reversed version of $s[n]$. Note that we can "time-reverse" the b_k coefficients rather than $x[n]$ and achieve the same result.

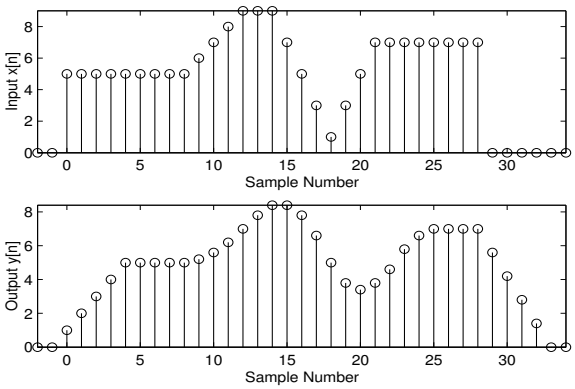


Figure 6.2: Input and output of a 5-point moving average filter.

That is, $b_k = (\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5})$, $M_1 = 0$, and $M_2 = 4$. We can think of the operation of this filter as replacing each sample of $x[n]$ with the average of that sample and the past four samples. As such, we often call filters like this *moving-average filters*. The result of this filtering for a particular signal is shown in Figure 6.2.

In this particular case, $x[n]$ has a support interval of $[0, 28]$ and is zero outside of this range. Consider what happens to the output signal, $y[n]$, near the edges of this range. First, the output sample at $y[0]$ will be dominated by zeros from outside of the support interval, because

$$y[0] = \frac{1}{5}x[n] + \frac{1}{5}0 + \frac{1}{5}0 + \frac{1}{5}0 + \frac{1}{5}0\,. \tag{6.7}$$

Similarly, $y[1]$, $y[2]$, $y[3]$, and $y[4]$ will also be affected by these zeros, but to a lesser extent. This effect can be seen in Figure 6.2 as $y[n]$ "ramps up" to the nominal values of $x[n]$. This effect is known as a *start-up transient*. A similar effect occurs beyond $y[28]$, where the signal takes a few samples to "die off". This effect is known as an *ending transient*. Both of these transients are known as *edge effects*, and need to be considered when filtering.

What do the edge effects do to the support length of the output signal? Well, from Figure 6.2 we can see that $y[n]$ has a support length four samples longer than that of $x[n]$. In general, the length of the output signal which is non-zero will be equal to the length of the input signal plus the order of the FIR filter.

There is one additional point that should be examined. Look at the location of the "dip" in Figure 6.2. In $x[n]$, the "dip" occurs at sample 18, but in $y[n]$ it occurs at sample 20. In fact, the entire support interval of $y[n]$ has not only gotten larger, it has also been shifted over to the right (or *delayed*) by two samples. Why is this? The delay introduced by this filter results from the fact that each output sample is an average of samples to its left. If we instead define this filter so that it considers two samples to both the right and left, we can eliminate this delay. That is, we define a different difference equation:

$$y[n] = \frac{1}{5}x[n+2] + \frac{1}{5}x[n+1] + \frac{1}{5}x[n] + \frac{1}{5}x[n-1] + \frac{1}{5}x[n-2]. \tag{6.8}$$

This modification, though, has taken a causal filter and made it non-causal.

Delay is a common problem for causal filters. In fact, the only causal filter that does not introduce delay is a zero-order amplifier system with a difference equation $y[n] = b_0 x[n]$. This system

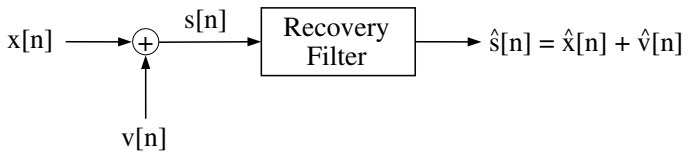


Figure 6.3: A block diagram of additive noise and a recovery filter that attempts to remove the noise.

changes the amplitude of a signal, but does nothing else. Compare this to the system with difference equation $y[n] = x[n - N]$. This system's only effect is to delay the signal by N samples. In some circumstances, the delay introduced by a causal filter does not affect the operation of the system. For our purposes in this laboratory, we will need to be careful to account for the delay introduced by FIR filters when comparing two signals with a mean-squared or RMS distortion measure.

6.2.3 Noise and distortion

One of the most common reasons to apply a filter is to attempt to remove *noise*. There is no single definition of noise, but the most general usage describes noise as any unwanted component of a signal. For instance, a common type of electrical noise has a sinusoidal characteristic with a frequency of 60 Hz. This noise arises from the frequency of the alternating current used to distribute electricity. This 60 Hz signal can "leak" into other systems and corrupt sensor measurements. Another common type of noise is *random noise*. This sort of noise typically has a jagged-looking characteristic. It typically manifests itself as static in audio signals and "snow" in images.

Filtering gives us a means of reducing the noise in a signal through *frequency blocking*. In general, filters operate by attenuating (i.e., blocking) certain frequencies in a signal while passing others with relatively little attenuation. Note that removing noise in this way requires the noise to have a different frequency-domain description than the signal of interest.

For instance, consider the example of 60 Hz sinusoidal noise. If our signal of interest is composed of frequencies above 60 Hz, we can treat this component as low frequency noise and attempt to remove it with a filter that blocks low frequencies. This sort of filter is generally called a *highpass filter*. If our signal has components above and below 60 Hz, we might try to remove the corrupting signal by only eliminating frequencies near 60 Hz. This would require a *bandpass filter*

Random noise typically has frequency components all over the spectrum. However, a good portion of these components will usually have higher frequency than the frequencies in our signal. Thus, we might consider the application of a *lowpass filter* that blocks high frequencies to reduce the amplitude of noise components.

Consider the block diagram in Figure 6.3. This is a model where a signal of interest, $x[n]$, is corrupted by the addition of a noise signal, $v[n]$. We apply a *recovery filter* to try to remove the noise component from $s[n] = x[n] + v[n]$. The resulting signal is $\hat{s}[n] = \hat{x}[n] + \hat{v}[n]$, where $\hat{x}[n]$ is the filtered signal of interest (which we hope will be as similar to $x[n]$ as possible) and $\hat{v}[n]$ is the filtered noise signal (which we hope will be as small as possible). Often we can tune the noise-removal filter to increase its "strength" (by, for instance, increasing the length of a moving average filter). The "stronger" the filter, the more noise we can eliminate. Unfortunately, the filter also distorts the signal of interest; a stronger filter will distort the signal of interest more. Thus, the use of filters to remove noise can be thought of as finding a tradeoff between two types of distortion. The goal, then, is to find the point where the total distortion (as measured by the mean-squared error or

RMS error between $x[n]$ and $\hat{s}[n]$) is minimized as a function of filter strength.

Nonlinear filtering

While standard FIR filters can be useful for noise reduction, in some cases we may find that they distort the desired signal too much. An alternative is to use *nonlinear* filters. Nonlinear filters have the potential to remove more noise while introducing less distortion to the desired signal; however, the effects of these filters are much more difficult to analyze.

Consider the case of an image, for instance. One of the most important features of images of natural scenes are *edges*. Edges in images are usually just sharp transitions where one object ends and another begins. If we are attempting to remove high-frequency noise from an image, we will often apply a lowpass filter. Edges, though, have considerable high-frequency content, so the edges in resulting image will be smoothed out. To get around this problem, we can consider the application of a common nonlinear filter called a *median filter*. Median filters replace each sample of a signal with the median (i.e., the most central value) of a block of samples around the original sample. That is, we can describe the operation of the median filter as

$$y[n] = Median(x[n + M_1], \ldots, x[n], \ldots, x[n - M_2]) \tag{6.9}$$

where

$$Median(x_1, \ldots, x_N) = \begin{cases} x_{((N+1)/2)} & N \text{ odd} \\ \frac{1}{2}(x_{(N/2)} + x_{(N/2+1)}) & N \text{ even} \end{cases} \tag{6.10}$$

and where $x_{(n)}$ is the n^{th} smallest of the values x_1 through x_N. The *order* of the median filter is given by $M_1 + M_2$, and it determines how many samples will be included in the median calculation. Note that the filter is noncausal because its output depends on future, as well as past and present, inputs. Unlike lowpass filters, median filters tend to preserve edges in signals very well. These filters are also very powerful for removing certain types of noise while introducing relatively little distortion. In this laboratory, we will examine the effect of applying nonlinear filters to two-dimensional signals.

6.2.4 Filtering two-dimensional signals

The above discussions of filtering are for one-dimensional signals. Suppose we would like to filter two-dimensional signals like images instead of just one-dimensional signals. There are three ways to approach this.

The first approach simply applies one-dimensional filters to each of the rows (or each of the columns) of an image. This approach tends to produce an "uneven" filtered signal that is, for instance, smoothed in one dimension but not the other. This unevenness is generally not desirable and motivates a second approach.

The second approach is somewhat "stronger" than the first. This approach applies one-dimensional filters to *both* the rows and the columns. In this lab we will adopt the convention that we first filter the columns, and then filter the rows of the resulting image. Most types of filters that we use in one dimension can be extended to two dimensions in this fashion. For example, if we apply the moving average filter with difference equation give in equation (6.6), for instance, this will have the effect of smoothing the image. Note that the edge effects and delay issues discussed earlier also apply to two-dimensional filtering done in this fashion.

If we apply that moving average filter with order 4 to the columns and then the rows of the image, what is the mathematical effect of the operation? It is not too difficult to see that each sample of the image has been replaced by the average of a 5×5 block of pixels. This suggests that we

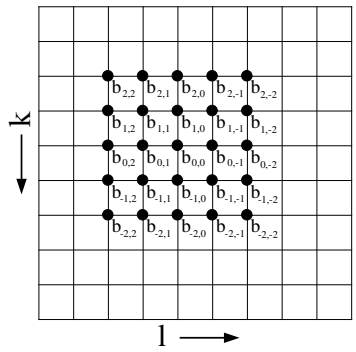


Figure 6.4: The coefficients of a two-dimensional moving average filter. In this figure, pixels exist at the intersection of the horizontal and vertical lines.

could describe this filtering operation in terms of two-dimensional set of filtering coefficients. For instance, the difference equation for this two-dimensional filter would be

$$y[m,n] = \sum_{k=0}^{4}\sum_{l=0}^{4} \frac{1}{25} x[m-k, n-l]. \tag{6.11}$$

This operation is equivalent to filtering with a two-dimensional set of coefficients $b_{k,l}$, where $b_{k,l} = \frac{1}{25}$ for $k = 0, \ldots, 4$ and $l = 0, \ldots, 4$.

This result suggests the third, most general, approach to FIR filtering of two-dimensional signals. The general difference equation for this approach is

$$y[m,n] = \sum_{k=-M_1}^{M_2}\sum_{l=-N_1}^{N_2} b_{k,l} x[m-k, n-l]. \tag{6.12}$$

$[-M_1, M_2]$ and $[-N_1, N_2]$ define the range of nonzero coefficients. Note that a filter, such as the one defined by equation 6.11, is *causal* if M_1 and N_1 are non-negative. However, we should also note that in image processing, causality is rarely important. Thus, two-dimensional FIR filters typically have coefficients centered around $b_{0,0}$. A schematic of such a set of filter coefficients is shown in Figure 6.4.

6.2.5 Image processing with FIR filters

If you've ever used photo editing software like Adobe Photoshop, you may have seen operations called "smoothing" and "sharpening". These and many similar operations are typically implemented using simple two-dimensional FIR filters. We will consider three such operations in this laboratory: smoothing (or *blur*), *edge finding*, and sharpening (or *edge enhancement*)

We've already suggested that a moving average filter performs a smoothing operation. However, there are more advanced ways of smoothing. Consider, for instance, a filter that weights samples nearby more strongly than those that are far away. This performs a "weaker" smoothing, but it also introduces less distortion. Because of this, these sorts of filters are often more useful for random noise reduction than standard moving average filters.

The "edge finding" filter highlights edges in an image by producing large positive or negative values while setting constant regions of the image to zero. The most basic edge finding filter is a simple one-dimensional *first difference* filter. A first difference filter has the difference equation

$$y[n] = x[n] - x[n-1]. \tag{6.13}$$

This filter will tend to respond positively to increases in the signal and negatively to decreases in the signal. Adjacent input samples that are identical (or nearly so), though, will tend to cancel one another, causing the output to be zero (or close to zero). There are various two-dimensional "equivalents" of the first-difference filter, many of which respond to edges of a particular orientation. One general edge-finding filter has the following difference equation:

$$\begin{array}{rcccccc} y[m,n] = & \frac{1}{4}x[m+1,n+1] & - & x[m+1,n] & + & \frac{1}{4}x[m+1,n-1] \\ & - \; x[m,n+1] & + & 3x[m,n] & - & x[m,n-1] \\ & + \; \frac{1}{4}x[m-1,n+1] & - & x[m-1,n] & + & \frac{1}{4}x[m-1,n-1] \end{array} \tag{6.14}$$

This filter "finds" edges of almost any orientation by outputting a value with large magnitude wherever an edge occurs. Both the first difference filter and this general edge-finding filter are examples of highpass filters. Note the "oscillatory" pattern of b_k values such that adjacent coefficients are negatives of one another. This pattern is characteristic of highpass filters. Note that both of these filters will typically produce both positive and negative values, even if our input signal is strictly non-negative. Also note that for both of these filters, the average of the b_k coefficients is zero; this means that these filters tend to "reject" constant regions of an input signal by setting them to zero.

The third operation, sharpening, makes use of an edge finding filter as well. Basically, the sharpening filter produces a weighted sum of the output of an edge-finding filter and the original image. Suppose that $x[m,n]$ is the original image, and $y[m,n]$ is the result of filtering $x[m,n]$ with the filter defined in equation (6.14). Then, the result of sharpening, $z[m,n]$ is given by

$$z[m,n] = x[m,n] + by[m,n], \tag{6.15}$$

where b controls the amount of sharpening; higher values of b produce a "sharper" image. Note that $z[m,n]$ can also be viewed as the output of a single filter. For display purposes, we will *threshold* the resulting signal so that the output image has the same range of data values as the input image. That is, assuming that our input image has values between 0 and 255, the final output of the sharpening operation, $\hat{z}[m,n]$ will be

$$\hat{z}[m,n] = \begin{cases} 0 & z[m,n] < 0 \\ z[m,n] & 0 \le z[m,n] \le 255 \\ 255 & 255 < z[m,n] \end{cases} \tag{6.16}$$

Note that thresholding is a *nonlinear* operation, but it is not crucial to the sharpening process. This final result can also be considered to be the output of a single nonlinear filter.

Sharpening is a useful operation when an image has undergone an undesired smoothing operation. This happens frequently in optical systems when they are not entirely in focus. Unlike smoothing filters, though, sharpening filters tend to enhance random noise; often they may make "noise-like" components of a signal visible where they were not visible before.

6.3 Some MATLAB commands for this lab

- **1-D Filtering in MATLAB:** The usual method for causal filtering in MATLAB is to use the `filter` command, like this:


```
>> yy = filter(bb,1,xx);
```

(We'll use the second parameter later in the course when we study IIR filters.) `xx` is a vector containing the discrete-time input signal to be filtered, `bb` is a vector of the b_k filter coefficients, and `yy` is the output signal. The first element of this vector, `bb(1)`, is assumed to be b_0.

By default, `filter` returns a portion of the filtered signal equal in length to `xx`. Specifically, the resulting signal includes the start-up transient but not the ending transient. This means that the output will be delayed by an amount determined by the coefficients of the filter.

A method for filtering which does not introduce delay is often desirable, i.e. a noncausal filtering method, especially when calculating RMS error between filtered and original versions of a signal. The command `filter2` is meant as a two-dimensional filtering routine, but it can be used for 1-D filtering as well. Further, it can be instructed to return a "delay-free" version of the output signal. When using `filter2`, it is important that `xx` and `bb` are either both row vectors or both column vectors. Then, we use the command

```
>> yy = filter2(bb,xx,'same');
```

where `xx` is the input signal vector, `yy` is the output signal vector, and `bb` is the vector of filter coefficients. If the length of the vector `bb` is odd, the b_0 coefficient is taken to be the coefficient at center of the vector `bb`. If the length of `bb` is even, b_0 is taken to be just left of the center. The output of `filter2` has support equal to that of the input signal `xx`.

Though we will not use these additional options, we can also have `filter2` return the full length of the filtered signal (the length of the input signal plus the order of the filter) like this:

```
>> yy = filter2(bb,xx,'full');
```

or just the portion not affected by edge effects (the length of the input signal minus twice the order of the filter), like this:

```
>> yy = filter2(bb,xx,'valid');
```

- **2-D Filtering in MATLAB:** Three approaches to filtering a two-dimensional signal were mentioned in Section 6.2.4. The first approach, which simply applies a one-dimensional filter to each row of the image (alternatively, to each column) can be implementing with the MATLAB commands described in the previous bullet.

  The second approach applies a one-dimensional filter first to the columns and then to the rows of the image produced by the first stage of filtering. If the one-dimensional filter is causal with coefficients b_k contained in the MATLAB vector `bb` and the image is contained in the 2-dimensional matrix `xx`, then this approach can be implemented with the command

  ```
  >> yy = filter(bb,1,filter(bb,1,xx)')';
  ```

  Note that we do not need to vectorize the image `xx`, because when presented with an matrix, `filter` applies one-dimensional filtering to each column. However, to perform the second stage of filtering (on rows of the image produced by the first stage), we need to transpose the image produced by the the first stage of filtering and then transpose the final result again to

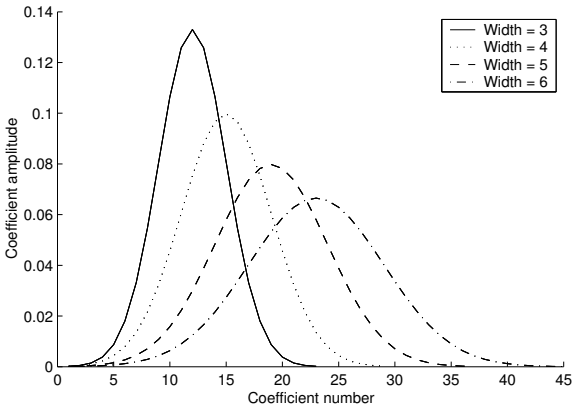


Figure 6.5: The coefficients for `g_smooth` filters with varying widths.

restore the original orientation. This approach will introduce edge effects at the top and on one side of the image; however the resulting image will be the same size as `x`.

The third approach uses a two-dimensional set of coefficients $b_{k,l}$. If these coefficients are contained in the matrix `bb` and the image is contained in the matrix `xx`, then the filter can be implemented with the command

```
>> yy = filter2(bb,xx,'same');
```

Note that the `'same'` parameter indicates that the filter is non-causal and thus the $b_{0,0}$ coefficient is located as near to the center of the matrix `bb` as possible. The same alternate third parameters for `filter2` that are listed in the 1-D filtering section apply here as well.

- **Generating filter coefficients:** We will be examining the effects of many types of filters in this laboratory. Some have filter coefficients that can be generated easily in MATLAB. Others require a function (which we will provide to you) to generate. Note that the the vectors representing the b_k's will be column vectors.

  1. An *L-point moving average* filter has filter coefficients given by `bb = ones(L,1)/L`.
  2. A *first-difference* filter has filter coefficients given by `bb = [1; -1];`
  3. The function `g_smooth` produces coefficients for a particular type of smoothing (low-pass) filters with easily tunable "strength". `g_smooth` takes a single real-valued parameter, which is the "width" of the filter, and returns a set of tapered filter coefficients of the corresponding filter, `bb`. For example,

     ```
     >> bb = g_smooth(1.2);
     ```

     returns the coefficients for a filter with "width" 1.2. A `g_smooth` filter with width 0 will pass the input signal without modification, and higher widths will smooth more strongly. Good nominal values for the width range from 0.5 to 2. The b_k coefficients for `g_smooth` filters with several widths are plotted in Figure 6.5.
  4. The function `g_smooth2` is the two-dimensional equivalent of `g_smooth`. It again takes a single input parameter (the width of the filter) and returns the two-dimensional set of filter coefficients of the corresponding filter. For example,


```
>> bb = g_smooth(0.8);
```

returns the coefficients of a filter with width 0.8.

5. In Section 6.2.5, we presented a general-purpose two-dimensional edge-finding filter in equation (6.14). The coefficients for this filter are given by

   ```
   >> bb = [.25, -1, .25; -1, 3, -1; .25, -1, .25];
   ```

6. In Section 6.2.5, we also discussed a method for implementing a sharpening filter. Since we include a threshold operation, this operation is nonlinear and cannot be accomplished using only an FIR filter. Thus, we provide the `sharpen` command, which takes an image and a sharpening "strength" and returns a sharpened image:

   ```
   >> yy = sharpen(xx,0.7);
   ```

   The second parameter is the strength factor, b, as discussed in Section 6.2.5. A sharpening strength of 0 passes the signal without modification.

7. As described in Section 6.2.3, median filters are a special type of nonlinear filter, and they cannot be described using linear difference equations. To use a median filter on a one-dimensional signal, we use the command[2] `medfilt1` like this:

   ```
   >> yy = medfilt1(xx,N);
   ```

   `N` is the *order* of the median filter, which simply describes how many samples we consider when taking the median. In two dimensions[3], we use `medfilt1` twice:

   ```
   >> yy = medfilt1(medfilt1(xx,N)',N)';
   ```

   Again, `N` is the order of the median filter. Here, we are using a one-dimensional filter on both the rows and columns of the image. Note that since `medfilt1` operates down the columns, we need to transpose the image between the filtering operations and again at the end.

6.4 Demonstrations in the Lab Section

- Filtering in MATLAB.
- FIR filters for noise reduction
- Image processing with FIR filters
- Median filtering

6.5 Laboratory Assignment

1. (Noise reduction in 1-D) In this problem, you investigate noise-reduction on one-dimensional signals. Download the file `lab6_data.mat`, which contains various signals for this lab. In this problem, we will consider the signal `simple`, which is a noise-free one-dimensional signal, and `simple_noise`, which is the same signal with corrupting random noise.

[2] `medfilt1` is a part of the signal processing toolbox.

[3] We can also use `medfilt2`, but this function is a part of the Image Processing Toolbox which we do not require for this course. `medfilt2` works by outputting the median of an $N \times N$ block of the image.

(a) (Effects of delay) First, we'll examine the delay introduced by the two filtering implementations, `filter` and `filter2`, that we will be using. Filter `simple` with a 7-point running average filter. Do this twice, first using `filter` and then using `filter2` with the `'same'` parameter[4].

- Use `subplot` and `plot` to plot the original signal and two filtered signals in three subplots of the same figure.
- One of the filtering commands has introduced some delay. Which one? How many samples of delay have been added?
- Compute the mean-squared error between the original signal and the two filtered signals. Which is lower? Why?

(b) (Measuring distortion in 1-D) Now, use `filter2` to apply the same 7-point running average filter to the signal `simple_noise`. Referring to Figure 6.3, we consider `simple` to be the signal of interest $x[n]$, `simple_noise` to be the noise corrupted signal $s[n]$, and their difference to be the noise, $v[n] = s[n] - x[n]$. Note that the lower of the two mean-squared errors that you computed in Problem 1a is $MS(\hat{x}[n] - x[n])$, which is a measure of the distortion of the signal of interest introduced by the filter.

- Compute the mean-squared error between `simple` and `simple_noise`. Referring back to Figure 6.3, this is $MS(v[n])$, the mean-squared value of the noise.
- Compute the mean-square error between your filtered signal and `simple`. This value is $MS(\hat{s}[n] - x[n])$, which is a measure of how a good a job the filter has done at recovering the signal of interest.
- Determine the distortion due to noise at the output of your reconstruction filter (i.e., $MS(\hat{v}[n])$) by subtracting $MS(\hat{x}[n] - x[n])$ from $MS(\hat{s}[n] - x[n])$.
- Compare $MS(\hat{v}[n])$ and $MS(\hat{s}[n] - x[n])$ to $MS(v[n])$. What is the dominant source of distortion in this filtered signal?

(c) (Running average filters in 1-D) Use `filter2` to apply a 3-point, a 5-point, and an 9-point moving average filter to `simple_noise`.

- Use `plot` and `subplot` to plot the original signal, the three filtered signals, and the three sets of filter coefficients, in seven panels of the the same figure.
- Compute the mean-squared error between each filtered signal and `simple`.
- Which of the four moving average filters that you have applied has the lowest mean-squared error? Compare this value to $MS(v[n])$.

(d) (Tapered smoothing filter in 1-D) Download the file `g_smooth.m`, and use it to generate filter coefficients with "widths" of 0.5, 0.75, and 1.0. (Note the lengths of the returned coefficient vectors. You should plot the filter coefficients to get a sense of how the "width" factor affects the them.) Use `filter2` to apply these filters to `simple_noise`.

- Use `plot` and `subplot` to plot the three filtered signals and the three sets of coefficients in six panels of the same figure.
- Compute the mean-squared error between each filtered signal and `simple`.
- Which of these filtered signal has the lowest mean-squared error? Compare this value to the lowest mean-squared error that you found for the moving average filters and to $MS(v[n])$.

[4]Henceforth, every time you use `filter2` in this laboratory, you should use the `'same'` parameter.

2. (Noise reduction on images) In this problem, you look at the effects applying smoothing filters to an image for noise reduction. Download the files `peppers.tif`[5] and `peppers_noise1.tif`. The first is a "noise-free" image, while the second is an image corrupted by random noise. Load these two images into MATLAB.

   (a) (Examining 2-D filter coefficients) We'll be using the function `g_smooth2` to produce filter coefficients for this problem. To get a sense of what these coefficients look like, generate the coefficients for a `g_smooth2` filter with width 5. In two side-by-side subplots of the same figure:

   - Display the coefficients as an image using `imagesc`.
   - Generate a surface plot of the coefficients using the command `surf(bb)` (assuming your coefficients matrix is called `bb`).

   (b) (Examine the effects of noise) First, we'll consider the noisy signal `peppers_noise1`.

   - Use `subplot` to display `peppers` and `peppers_noise1` side-by-side in a single figure. Remember to set the color map, set the axis shape, and include a colorbar as you did in lab 4.
   - Compute the mean-squared error between these two images.

   (c) (Minimizing the MSE) Our goal is to find a `g_smooth2` reconstruction filter that minimizes the mean-squared error between the filtered image and the original, noise-free image. Use `filter2` when filtering signals in this problem.

   - Find a filter width that minimizes the mean-squared error. What is this filter width and the corresponding mean-squared error? (Hint: you might want to plot the mean-squared error as a function of filter width.)
   - Display the filtered image with the smallest mean-squared error.
   - Look at some filtered images with different widths. Can you find one that looks better than the minimum mean-squared error image[6]? What filter width produced that image?

3. (Salt and pepper noise in images) Next, we'll look at methods of removing a different type of random noise from this image. Download the file `peppers_noise2.tif` and load it into MATLAB. This signal is corrupted with *salt and pepper* noise, which may result from a communication system that loses pixels.

   (a) (Examining the noise) First, let's see what we're up against. Salt and pepper noise randomly replaces pixels with a value of either 0 or 255. In this image, one-fifth of the pixels have been lost in this manner.

   - Display `peppers_noise2`.
   - Compute the mean-squared error between this image and `peppers`.

   (b) (Using lowpass filters) Now, let's try using some `g_smooth2` filters to eliminate this noise. Start by using `filter2` to filter `peppers_noise2` with a `g_smooth2` filter of width 1.3. Note that this is very close to the optimal width value.

   - Display the resulting image.

[5] Like "cameraman", "peppers" is a standard image used for testing image processing routines. Our version, however, is smaller than the traditionally used image.

[6] Though mean-squared error is widely used as a measure of signal distortion, it is well known that its judgments of quality do not always correspond closely to the eye's judgments of quality.

- Compute the mean-squared error.

(c) (Using median filters) Finally, let's use a median filter to try to remove this noise. Apply median filters of order 3 and 5 to `peppers_noise2`.

- Use `subplot` to display the two filtered images side-by-side in the same figure.
- Compute the mean-squared errors between the median-filtered signals and `peppers`.
- Look at the filtered images and describe the distortion that the median filters introduce into the signal.
- Compare the median filter to the `g_smooth2` filters. Discuss both measured distortion and the appearance of the filtered signals.

4. (Edge-finding and enhancing) In this last problem, we'll look at edge-finding and sharpening filters.

(a) (Applying a first difference filter) In order to see how edge-finding filters work, let's start in one dimension. Use `filter` to apply a one-dimensional first difference filter to the signal `simple` (which can be found in `lab6_data.mat`).

- Plot the resulting signal.
- There are five non-zero "features" of this signal. (These features should be clear from the plot.) Describe them and what they correspond to in `simple`.

(b) ("Finding" edges) Now we'd like to look at the effects of the general edge-finding filter presented in Section 6.2.5. Use `filter2` to apply this filter to `peppers`.

- Display the resulting image.
- Describe the resulting image.
- Zoom in on the filtered image and examine some of the more prominent edges. What do you notice about these edges? (Hint: Are they just a "ridge" of a single color?)

(c) (Sharpening an image) Download `sharpen.m` and use the function to display several sharpened versions of the `peppers` image.

- Display the sharpened image with a "strength" of 1 alongside the original `peppers` image using `subplot`.
- Zoom in on this sharpened image. What makes it look "sharper"? (Hint: Again, look at the prominent edges of the images. What do you notice?)
- The sharpened images (especially for strengths greater than 1) generally appear more "noisy" than the original image. Speculate as to why this might be the case.

(d) (Using sharpening to remove smoothing) Finally, we want to try using the "sharpen" function to undo a blurring operation. Download the file `peppers_blur.tif` and load it into MATLAB.

- Compute the RMS error between `peppers` and `peppers_blur`.
- Use sharpen to "de-blur" the blurred image. Find the sharpening strength that minimizes the RMS error of the "de-blurred" image. Include this strength and its corresponding RMS error in your report.
- Display the "de-blurred" image with the minimum RMS error and alongside `peppers_blur` using `subplot`. Include the resulting figure in your report.

Note that sharpening is very much a perceptual operation. The minimum distortion sharpened image may not look terribly much improved. Look at what happens as you increase the sharpening factor even more. With additional "sharpening," the (measured) distortion may increase, but the result looks better perceptually.

5. On the front page of your report, please provide an estimate of the average amount of time spent outside of lab by each member of the group.

Laboratory 7

Decoding DTMF: Filters in the Frequency Domain

7.1 Introduction

In Lab 6, you examined the behavior of several different filters. Some of the filters were "smoothing filters" that averaged the signal over many samples. Others were "sharpening" filters that accentuated transitions and edges. While it is very useful to understand the effects of these filters in the *time-domain* or (for images) the *spatial-domain*, it is often not easy to quantify these effects, especially when we are dealing with more complicated filters. Thus, just as we did with signals, we would like to obtain a better understanding of the behavior of our filters in the frequency-domain.

Assuming that our filter is linear and time-invariant, we can talk about the filter having a *frequency response*. We derive the frequency response in the following way. We know that if we put a complex exponential signal into such a filter, the output will be a scaled and shifted complex exponential signal with the same frequency. The amount of scaling and phase shift, though, is dependent on the frequency of the input signal. If we send a complex exponential signals with some frequency through the filter, we can measure the scaling and phase shifting of that signal. The collection of complex numbers which corresponds to this scaling and shifting for all possible frequencies is known as the filter's *frequency response*. The magnitude of the frequency response at a given frequency is the filter's *gain* at that frequency.

In this lab, we will be using the frequency response of filters to examine the problem solved by *telephone touch-tone dialing*. The problem is this: given a noisy audio channel (like a telephone connection), how can we reliably transmit and detect phone numbers? The solution, which was developed at AT&T, involves the transmission of a sum of sinusoids with particular frequencies. In order for this solution to be feasible, we must be able to easily decode the resulting signal to determine which numbers were dialed. We will see that we can do this easily by considering filters in the frequency domain.

7.1.1 "The Question"

- How can we decode telephone touch-tone (DTMF) signals?

7.2 Background

7.2.1 DTMF signals and Touch Tone™ Dialing

Whenever you hit a number on a telephone touch pad, a unique tone is generated. Each tone is actually a sum of two sinusoids, and the resulting signal is called a *dual-tone multifrequency* (or *DTMF*) signal. Table 7.1 shows the frequencies generated for each button. For instance, if the "6" button is pressed, the telephone will generate a signal which is the sum of a 1336 Hz and a 770 Hz sinusoid.

Frequencies	1209 Hz	1336 Hz	1477 Hz
697 Hz	1	2	3
770 Hz	4	5	6
852 Hz	7	8	9
941 Hz	*	0	#

Table 7.1: DTMF encoding table for touch tone dialing. When any key is pressed, the tones of the corresponding row and column are generated.

We will call the set of all seven frequencies listed in this table the *DTMF frequencies*. These frequencies were chosen to minimize the effects of signal distortions. Notice that none of the DTMF frequencies is a multiple of another. We will see what happens when the signal is distorted and why this property is important.

Looking at a DTMF signal in the time domain does not tell us very much, but there is a common signal processing tool that we can use to view a more useful picture of the DTMF signal. The *spectrogram* is a tool that allows us to see the frequency properties of a signal as they change over time. The spectrogram works by taking multiple DFTs over small, overlapping segments[1] of a signal. The magnitudes of the resulting DFTs are then combined into a matrix and displayed as an image. Figure 7.1 shows the spectrogram of a DTMF signal. Time is shown along the x-axis and frequency along the y-axis. Note the bars, each of which represents a sinusoid of a particular frequency existing over some time period. At each time, there are two bars which indicate the presence of the two sinusoids that make up the DTMF tone. From this display, we can actually identify the number that has been dialed; you will be asked to do this in the lab assignment.

7.2.2 Decoding DTMF Signals

There a number of steps to perform when decoding DTMF signals. The first two steps allow us to determine the strength of the signal at each of the DTMF frequencies. We first employ a bank of bandpass filters with center frequencies at each of the DTMF frequencies. Then, we process the output of each bandpass filter to give us an indication of the strength of each filter's output. The third step is to "detect and decode." From the filter output strengths, we detect whether or not a DTMF signal is present. If it is not, we refrain from decoding the signal until a tone is detected. Otherwise, we select the two filters with the largest output strengths and use this information to determine which key was pressed. A block diagram of the DTMF decoder can be seen in Figure 7.2.

[1] Note that each segment is some very small fraction of a second, and the segments usually overlap by 25-75%.

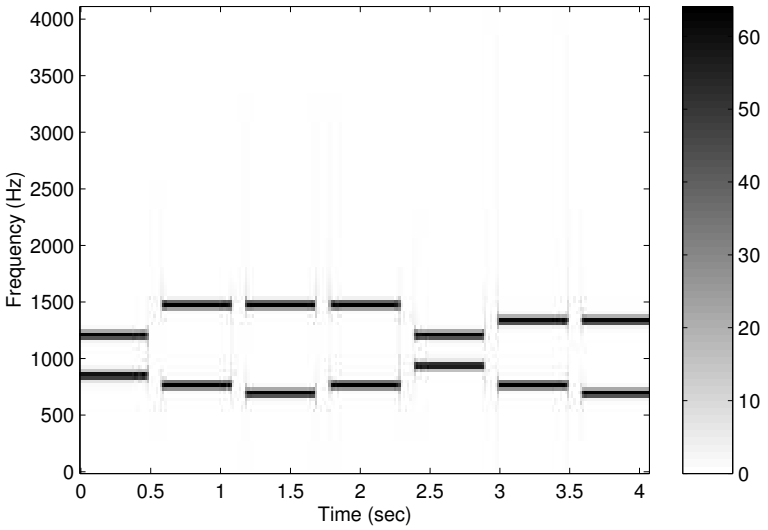


Figure 7.1: A spectrogram of a DTMF signal. Each horizontal bar indicates a sinusoid that exists over some time period.

Step 1: Bandpass Filters

From Lab 2, you may recall that correlating two signals provides us with a measure of how similar those two signals are. Since convolution is just "correlation with a time reversal," we can use this same idea to design a filter that passes a given frequency. If our filter's impulse response "looks like" the signal we want to pass, we should get a large amplitude signal out; similarly, signals that are different will produce smaller output signals.

When performing DTMF decoding, we want filters that pass only one of the DTMF frequencies and reject all of the rest. We can make use of the correlation idea above to develop such a *bandpass filter*. We want our impulse response to be similar to a signal with the frequency that we wish to pass; this is the filter's *center frequency*. This means that for a bandpass filter with center frequency f, we want our impulse response, h, to be equal to

$$h[k] = \begin{cases} \sin(2\pi f_c k/f_s) & 0 \leq k \leq M \\ 0 & else \end{cases} \tag{7.1}$$

From this equation, we have an FIR filter with order M. (Note that the support length of the impulse response is $M + 1$.) What should M be? M is a design parameter. You may remember from Lab 3 that correlating over a long time produces better estimates of similarity. Thus, we should get better differentiation between passed frequencies and rejected frequencies if M is large. There is a tradeoff, though. The longer M is, the more computation that is required to perform the convolution. Thus for efficiency reasons we would like M to be as small as possible. More computation also equates to more expensive devices, so we prefer smaller M for reasons of device economy as well. Since we have seven DTMF frequencies, we will also have seven bandpass filters in our system; in our decoder system, we will choose a different value of M for each bandpass filter.

Because of the relatively small set of frequencies of concern in DTMF decoding, we will see that larger M do not necessarily produce better frequency differentiation. In order to judge how good a bandpass filter is at rejecting unwanted DTMF frequencies, we will define the *gain-ratio*, R. Given

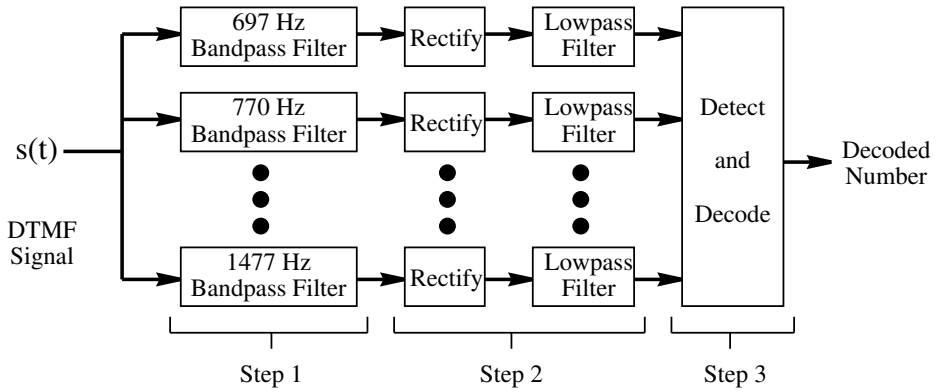


Figure 7.2: A block diagram of the DTMF decoder system. The input is a DTMF signal, and the output is a string of numbers corresponding to the original signal.

a filter with center frequency f_c and frequency response $\mathcal{H}$, the gain-ratio is

$$R = \frac{|\mathcal{H}(f_c)|}{\max|\mathcal{H}(\hat{f})|} \tag{7.2}$$

where $\hat{f}$ is in the set of DTMF frequencies and $\hat{f} \neq f_c$. In words, we define R to be the ratio of the filter's gain at its center frequency to the *next-highest* gain at one of the DTMF frequencies. Having a high gain-ratio is desirable, since it indicates that the filter is rejecting the other possible frequencies.

Note that since we will be comparing the outputs of a variety of bandpass filters, we also need to normalize each filter by the center frequency gain. Thus, we will need to record not only the M that we select but also the center frequency gain. You will be directed to record and include these gains in the lab assignment.

Step 2: Determining filter output strengths

In order to measure the strength of the filter's output, we actually want to measure (or follow) the envelope of the filter outputs. To follow just the positive envelope of the signal, we first need to eliminate the negative portions of the signal. If we simply truncate all parts of the signal below zero, we have applied a *half-wave rectifier*. Alternately, we can simply take the absolute value of the signal, in which case we have applied a *full-wave rectifier*. It is possible to build rectifiers using diodes, and it turns out that half-wave rectifiers are easier to design. However, full-wave rectifiers are preferable, and they are no more difficult to implement in MATLAB. Thus, we will use full-wave rectifiers[2]. See Figure 7.3 to see the effects of these two types of rectifiers.

If we now pass the rectified signal through a smoothing filter, the output will be a nearly constant signal whose value is a measure of the strength of the filter's input at the center frequency of the filter. To accomplish this smoothing we will use a simple moving average filter with impulse response

$$h_{LP} = \begin{cases} \frac{1}{M_{LP}+1} & 0 \leq k \leq M_{LP} \\ 0 & else \end{cases} \tag{7.3}$$

[2]Note that the names "full-wave rectifier" and "half-wave rectifier" come from the circuit implementation of these systems

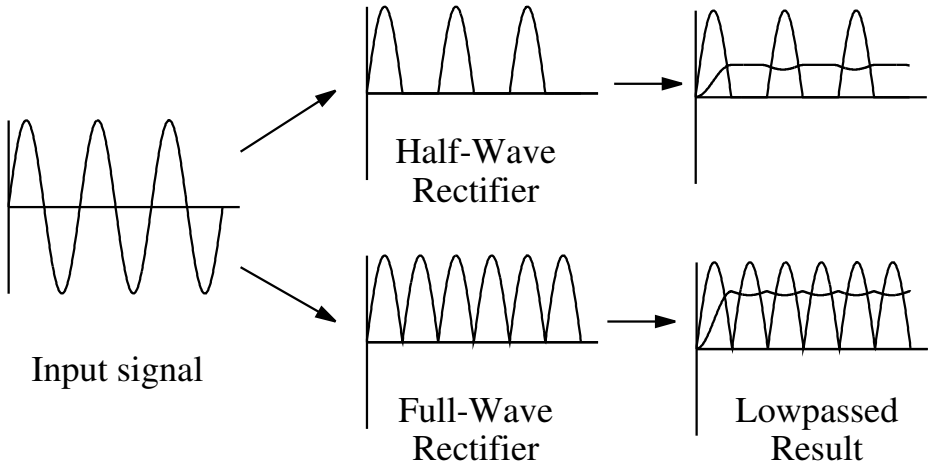


Figure 7.3: A comparison of half-wave and full-wave rectification. Notice that full-wave rectification allows us to achieve a higher output signal level after lowpass filtering.

The order of this filter is M_{LP}. The value M_{LP} (and thus the corresponding strength of the smoothing filter) is a design parameter of the decoder system. When choosing M_{LP}, there is a tradeoff between amount of smoothing and transient effects. If our filter's impulse response is not long enough, the output signal will still have significant variations. If it is too long, transient effects will dominate the output of the filter. If it is too short, the system may "smooth over" short DTMF tones or periods of silence. Note that in our decoder system, we will apply the same smoothing filter to the output of each filter. Figure 7.3 shows the results of smoothing for half-wave and full-wave rectified signals.

Step 3: "Detect and Decode"

Once we have processed the outputs of the bandpass filters, we can now detect whether or not a DTMF tone is present and, if it is, determine which key was pressed to produce it. Ultimately, we want to convert our signal into a sequence of keys pressed to produce this sequence. The detect-and-decode step itself involves three steps.

The first step is to detect whether a DTMF tone is actually present at a particular time. If it is not, we risk making an error in our decoding of the input signal. We detect the presence of a DTMF tone by comparing the rectified and smoothed bandpass filter outputs to a threshold, c. If any of the signals are greater than the threshold, then we decide that a DTMF tone is present. Figure 7.4a shows the rectified-and-smoothed output from one of the bandpass filters and the threshold to which it is compared. The threshold is a design parameter of the decoder. We generally want to the threshold to be high enough that noise will not "trigger" the detector during a period of silence, but low enough that noise won't pull the signal from a DTMF tone below the threshold. Figure 7.4b shows a noisy DTMF tone with the threshold.

When the input signal is noisy, there is an additional problem during the transient portions at the beginning and end of a DTMF tone. Near the threshold crossing, the noise could cause the signal to cross the threshold several times, as shown in Figure 7.4c; this might cause a single DTMF tone to be decoded as multiple key presses. To avoid this problem, we do not make a detection decision for

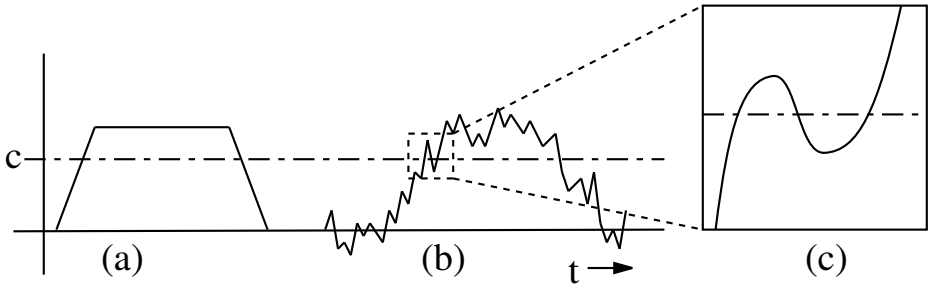


Figure 7.4: An illustration of the detector subsystem. (a) A clean DTMF signal is compared to a threshold, c. (b) The threshold should be set so that noise will not produce false tone detections or miss true tone detections in the presence of noise. (c) Near the threshold crossing, noise can cause multiple detections.

every sample of the input signal. Instead, we only make a decision every 100 samples. This makes it more likely that there will only be one decision made in the vicinity of the threshold crossing. It also reduces computation time somewhat. Note that the number 100 is somewhat arbitrary. We can choose a smaller number, but then we increase the risk the multiple-crossing problem. Alternatively, we can make it larger; however, it we make it too large, our detector may miss short tones or silences.

The second step is to decode of the DTMF tones that we have detected in the previous step. By "decode," we simply mean that we must decide which key was pressed to generate a particular DTMF tone. To do this, we determine which two bandpass filters have the largest output at each time when a DTMF tone was detected. Then, we effectively perform a table look-up to see which key was pressed at these times. The result is a sequence of decoded numbers corresponding to key presses. However, each DTMF tone will generally produce a sequence of identical numbers since it is "decoded" at many times during the duration of the DTMF tone. To translate this sequence of numbers into a sequence of key presses, we need a third step.

The third step simply combines adjacent, identical numbers in the decoded sequence. That is, a "run" of identical numbers is replaced by a single number. Through this process, each DTMF tone is finally represented by a single number. Note that for this process to work correctly, our sequence of numbers must also contain an indication of when no tone was present. Otherwise, any repeated key press would be decoded as only a single key press.

7.2.3 Decoder Robustness

Whenever designing a communication system, like the DTMF coder/decoder described here, it is important to consider how the system behaves in the presence of undesirable effects. For instance, the telephone system could corrupt our DTMF signal with some amount of static. Under such conditions, how well would the decoder work? How much noise can the system tolerate? These are all questions about the *robustness* of the decoder system to noise. No system can work perfectly under less than ideal conditions, so it is important to understand when and how a system will fail. In the lab assignment, we will examine the robustness of this system under noise.

7.2.4 Sidenote: Searching Parameter Spaces

Quite frequently, you will find yourself in the position of searching for a "good value" for a particular parameter about which you have no other information. In these cases, there are some techniques that we can employ to speed the search. The basic idea is that we want to get "in the ballpark" before we worry about finding locally optimum solutions. To do this, we think about varying parameters over factors of 2 or factors of 10. Thus, you might try parameter values of 0.01, 0.1, 1, 10, and 100 to get a general notion of how the system responds to a parameter. Once we have done this, we can then isolate a smaller range over which to optimize. This prevents us from spending too much time searching aimlessly.

7.3 Some MATLAB commands for this lab

- **Computing the frequency response of an FIR filter**: The MATLAB command `freqz` returns the frequency response of a filter at a specified number of discrete-time frequencies. The general usage of `freqz` for causal FIR filters is

  ```
  >> [H,w] = freqz(bb,1,n);
  ```

  Here, `bb` is the set of filter coefficients (i.e., the impulse response) of the FIR filter, n is the number of points in the range $[0, \pi)$ at which to evaluate the frequency response, `H` is the frequency response, and `w` is the set of n corresponding discrete-time frequencies, which are spaced uniformly from 0 to π. The frequency response, `H`, is a vector of complex numbers which define the *gain* (`abs(H)`) and *phase-shift* (`angle(H)`) of the filter at the given frequencies.

  Alternatively, we can evaluate the frequency response only at a specified set of frequencies by replacing `n` with a vector of discrete-time frequencies. Thus, the command

  ```
  >> H = freqz(bb,1,[pi/3, pi/2, 2*pi/3]);
  ```

  returns the frequency response at the discrete-time frequencies $\frac{\pi}{3}$, $\frac{\pi}{2}$, and $\frac{2\pi}{3}$.

  When we apply a filter to a sampled signal with sampling frequency `fs` (in samples per second), we can evaluate the frequency response at the discrete-time frequencies corresponding to a specified set of continuous time-frequencies in Hertz in the following manner:

  ```
  >> H = freqz(bb,1,[100 200 400 500]/fs*2*pi);
  ```

  This converts the specified continuous-time frequencies into discrete-time frequencies and evaluates the frequency response at those points.

- **Sorting a vector:** The MATLAB command `sort` sorts a vector in ascending order. Thus, given a vector `x`, the command

  ```
  >> y = sort(x);
  ```

  produces a vector `y` such that `y(1)` is the smallest value in `x` and `y(end)` is the largest value in `x`.

- **Creating matrices of ones and zeros:** In order to create arrays of arbitrary size containing only ones or only zeros, we use the MATLAB `ones` and `zeros` commands. Both commands take the same set of input parameters. If only one input parameter is used, a square matrix with the specified number of rows and columns is generated. For instance, the command

  ```
  >> x = ones(5);
  ```

  produces a 5×5 matrix of ones. Two parameters specify the desired number of rows and columns in the matrix. For instance, the command

  ```
  >> x = zeros(4, 8);
  ```

  produces a 4×8 matrix (i.e., four rows and eight columns) containing only zeros. To generate column vectors or row vectors, we set the first or second parameter to 1, respectively.

- **The DTMF Dialer:** `dtmf_dial.m` is a DTMF "dialer" function. It takes a vector of key presses (i.e., a phone number) and produces the corresponding audio DTMF signal. *Note that this function as provided is incomplete; you will be directed to complete it in the laboratory assignment.* (The lines of code that you need to complete are marked with a `?`.) To produce the DTMF signal that lets you dial the number 555-2198, use the command:

  ```
  >> signal = dtmf_dial([5 5 5 2 1 9 8]);
  ```

  An optional second parameter will cause the function to display a spectrogram of the resulting DTMF signal:

  ```
  >> signal = dtmf_dial([5 5 5 2 1 9 8],1);
  ```

  This function assumes a sampling frequency of 8192 samples per second. Each DTMF tone has a length of 1/2 second, and the tones are separated by 1/10 second of silence. Note that the number 10 corresponds to a `'#'`, 11 corresponds to a `'0'`, and 12 corresponds to a `'*'`.

- **The DTMF Decoder:** `dtmf_decode.m` is an (incomplete) DTMF decoder function. (Once again, the lines of code that you need to complete are marked with a `?`.) It takes a DTMF signal (as generated by `dtmf_dial`) and returns the sequence of key-presses used to create the signal. Thus, if our DTMF signal is stored in `signal`, we decode the signal using the command:

  ```
  >> decoded = dtmf_decode(signal);
  ```

  An optional second parameter will cause the function to display a plot of the smoothed and rectified outputs of each bandpass filter:

  ```
  >> decoded = dtmf_decode(signal,1);
  ```

- **Bandpass Filter Characterization:** `dtfm_filt_char.m` is a function that we will use to help us calculate gain-ratios for the bandpass filters used in the DTMF decoder. We use the function to focus on one of the bandpass filters at a time. The function takes two parameters: the order, `M`, of one of the bandpass filter's impulse responses and the center frequency in Hertz, `frq`, of that filter. The function returns a vector containing the gain (i.e., the magnitude of the frequency response) at each of the DTMF frequencies, from lowest to highest. It also produces a plot of the frequency response with locations of the DTMF frequencies indicated. Use the following command to execute the function:


```
>> gains = dtmf_filt_char(M,frq);
```

A second optional parameter lets you suppress the plot:

```
>> gains = dtmf_filt_char(M,frq,0);
```

- **Testing the robustness of the DTMF decoder:** `dtmf_attack.m` is a function that tests the DTMF decoder in the presence of random noise. This function generates a standard seven digit DTMF signal, adds a specified amount of noise to the signal, and then passes it through your completed `dtmf_decode` function. The decoded string of key presses is compared to those that generated the signal. Since the noise is random, this procedure is repeated ten times. The function then outputs the fraction of trials decoded successfully. The function also displays the plot from the last execution of `dtmf_decode`. (Note: since each call to `dtmf_decode` takes a little time, this function is rather slow. Be patient with it.)

  For instance, to test the system with a noise power of 2.5, we use the following command:

  ```
  >> success_rate = dtmf_attack(2.5);
  ```

  The result is a number that provides the fraction of the 10 trials that were successful.

  Note that `dtmf_attack` is a complete function, but it calls both `dtmf_dial` and `dtmf_decode`, each of which you must complete.

7.4 Demonstrations in the Lab Section

- Examining the frequency response of FIR filters
- Dual tone multi-frequency signals
- Generating "synthetic" DTMF signals.
- Bandpass filters
- The DTMF decoder
- Noise and the DTMF decoder

7.5 Laboratory Assignment

1. (The DTMF dialer.) Before we can decode a DTMF signal, we need to be able to produce DTMF signals. In this problem, we'll write a function that takes a phone number and produces the corresponding DTMF signal, just like the telephone would produce if you dial the number.

   Download the function `dtmf_dial.m`, which is a nearly complete dialer function. You simply need to replace the question marks by code that completes the function. The first missing line of code generates a DTMF tone for each number in the input and appends it to the output signal. The second line of code appends a short silence to the signal to separate adjacent DTMF tones.

   - Complete the function and include the code in your lab report.

- Using your newly completed dialer function, execute the following command to create a DTMF signal and display it's spectrogram:

  ```
  >> signal = dtmf_dial([1 2 3 4 5 6 7 8 9 10 11 12],1);
  ```

  Include the resulting figure in your report. Note how each key press produces a different pattern on the spectrogram.

- What is the phone number that has been dialed in Figure 7.1?

2. (The bandpass filters of the DTMF Decoder.) As we have noted, a key part of the DTMF decoder is the bank of bandpass filters that is used to detect the presence of sinusoids at the DTMF frequencies. We have specified a general form for the bandpass filters, but we still need to choose the filter orders and create their impulse responses. In this problem you will be identifying good values for M.

   (a) (The impulse response of one bandpass filter.) First, we need to be able to create the impulse response for a bandpass filter. Using equation (7.1) with a sampling frequency f_s = 8192 Hz and $M = 50$, use MATLAB to create a vector containing the impulse response, `h`, of a 770 Hz bandpass filter[3].

   - What is the command that you used to create this impulse response?
   - Use `stem` to plot your impulse response.

   (b) (The frequency response of one bandpass filter.) When we talk about the response of a filter to a particular frequency, we can think about filtering a unit amplitude sinusoid with that frequency and measuring the amplitude and phase shift of the resulting signal. We can certainly do this in MATLAB, but it's far simpler to use the `freqz` command. Here, you'll use `freqz` to examine the frequency response and gain-ratio of a bandpass filter like the ones we'll use in the DTMF decoder.

   - Use `freqz` to calculate the frequency response of your 770 Hz bandpass filter at all seven of the DTMF frequencies[4]. Calculate the gain at each frequency, and include these numbers in your report.
   - From the frequency response of your filter at these frequencies, calculate the gain-ratio, R.
   - Do you think that this is a good gain-ratio for our bandpass filters? (Hint: You might want to come back to this problem after you've worked the remainder of this problem.)

   (c) (Choosing M for this bandpass filter.) Now, we'd like to see what happens when we change M for your 770 Hz bandpass filter. We've provided you with a function that will facilitate this. Download the file `dtmf_filt_char.m`. This function will help you to visualize the frequency response of these filters and to determine their gain at the DTMF frequencies.

   - Use this function to verify that the gains you calculated in Problem 2b were correct.
   - Include the frequency-response plot that `dtmf_filt_char` produces in your report.

[3] Remember that if a filter has order M, the support length of the impulse response should be $M + 1$.
[4] Remember that our system uses a sampling frequency of 8192 Hz

- The frequency response of this filter is characterized by several "humps" which are typically called *lobes*. Describe the frequency response in terms of such lobes. Vary M and examine the plots that result (you do not need to include these plots). Describe the differences in the frequency response as M (which represents the length of the filter's impulse response) is changed.
- What happens to the relative heights of adjacent lobes as M is changed?
- What features of the filter's frequency response contribute to the gain ratio R?
- For what values of M do we achieve gain ratios greater than 10?

(d) (A function for computing gain ratios.) You'll need to compute the gain-ratio repeatedly while finding good design parameters for the bandpass filters, so in this problem you'll automate this task. Write a function that accepts a vector of gains (such as that returned by `dtmf_filt_char`) and computes the gain ratio, R. (Hint: This is a simple function if you use the `sort` command. You can assume that the center frequency gain is the largest value in the vector of gains.)

- Include the code for this function in your report.

(e) (Specifying the bandpass filters.) For each bandpass filter that corresponds to one of the seven DTMF frequencies, we want to find a choice of M that yields a good gain ratio but also minimizes the computation required for filtering.

To do this, for each bandpass filter frequency, use `dtmf_filt_char` and your function from Problem 2d to calculate R for all M between 1 and 200. Then, plot R as a function of M. You can save some computation time by setting the third parameter of `dtmf_filt_char` to zero to suppress plotting. You should be able to identify at least one local maximum[5] of R on the plot. The "optimal" value of M that we are looking for is the smallest one that produces a local maximum of R that is greater than 10.

- Create this plot of R as a function of M for the bandpass filter with a center frequency of 770 Hz. Include the resulting plot in your report.
- Identify the "optimal" value of M for this filter, the associated center frequency gain, and the resulting value of R.
- Repeat the above two steps for the remaining six bandpass filters. (You do not need to include the additional plots in your report.) Create a table in which you record the center frequency, the optimal M value, the associated center frequency gain, and the resulting value of R.

3. (Completing the DTMF decoder.) Now we have designed the bank of bandpass filters that we need for the DTMF decoder. In this problem, we'll use the parameters that we found to help us complete the decoder design. Download the file `dtmf_decode.m`. This function is a nearly complete implementation of the DTMF decoder system described earlier in this lab. There are several things that you need to add to the function.

(a) (Setting the M's and the gains of the bandpass filters.) First, you need to record your "optimized" values of M and the center frequency gains in the function. Replace the question marks on line 29 by a vector of your optimized values of M. They should be in order from smallest frequency to largest frequency. Do the same on line 32 for the variable `G`, which contains the center frequency gains.

[5] A *local maximum* is basically just a point on the plot that is larger than all other values in its vicinity. It may or may not be the highest possible peak, which is called the *global maximum*.

- Make these modifications to the code. (At the end of this problem, make sure that you include your completed function in your report.)

(b) (Setting the impulse responses of the bandpass filters.) Also, you need to define the impulse response for each bandpass filter on line 49. Use equation (7.1) for this, where the filter's order is given by `M(i)`.

- Make this modifications to the code.

(c) (Selecting the order of the post-rectifier smoothing filter.) Next, you need to specify the post-rectifier smoothing filter, `h_smooth`. Temporarily set both `h_smooth` (line 36) and `threshold` (line 40) equal to 1 and run `dtmf_decode` on the DTMF signal you generated in Problem 1. This function displays a figure containing the rectified and smoothed outputs for each bandpass filter. With `h_smooth` equal to 1, no smoothing is done and we only see the results of the rectifier in this figure. We will use moving average filters of order M_{LP}, as defined by the MATLAB command

```
>> h_smooth = ones(M_LP+1,1)/(M_LP+1);
```

We want the smoothed output to be effectively constant during most of the duration of the DTMF tones, but we don't want to smooth so much that we might miss short DTMF tones or pauses between tones.

- Examine the behavior of the smoothed signal when you replace line 36 with moving average filters with order M_{LP} equal to 20, 200, and 2000. Which filter order, M_{LP} gives us the best tradeoff between transient effects and smoothing?
- Set `h_smooth` to be the filter you have just selected.

(d) (Detection threshold.) Finally, you need to identify a good value for `threshold`. `threshold` determines when our system detects the presence of a DTMF signal. `dtmf_decode` plots the threshold on its figure as a black dotted line. We want the threshold to be smaller than the large amplitude signals during the steady-state portions of a DTMF signal, but larger than the signals during the start-up transients for each DTMF tone. (Hint: When choosing a threshold, consider what might happen if we add noise to the input signal.)

- By looking at the figure produced by `dtmf_decode`, what would be a reasonable threshold value? Why did you choose this value?
- Set `threshold` to the value you have just selected.
- Now, execute `dtmf_decode` and include the resulting plot in your report. (Note: You can include this plot in black and white, if you like.)
- `dtmf_decode` should output the same vector of "key presses" that was used to produce your signal. What "key presses" does the function produce? Do these match the ones used to generate the DTMF signal? If not, you've probably made a poor choice of threshold.

(e) Remember to include the code for your completed `dtmf_decode` function in your report.

4. (Robustness of the DTMF decoder to noise.) In the introduction to this lab, we indicated that we would be transmitting our DTMF signals over a noisy audio channel. So far, though, we have assumed that the decoder sees a perfect DTMF signal. In this problem, we will examine the effects of additive noise on the DTMF decoder.

(a) Download the file `dtmf_attack.m`. Execute `dtmf_attack` with various noise powers. Find a value of noise power for which some but not all of the trials fail.

- What value of noise power did you find? (Hint: use the parameter searching method discussed in the background section to speed your search).
- Make a plot of the fraction of successes versus noise power. Include at least 10 values on your plot. Make sure that your minimum noise power has a success rate at (or at least near) 1 and your maximum noise power has a success rate at (or near) 0. Try to get a good plot of the transition between high success rates and low success rates. While making this plot, pay attention to the types of errors that the decoder is making.

(b) By examining the plots for failure trials and the types of errors that the decoder is making, you should be able to speculate about the source of the errors.

- What types of errors is the system making when it decodes the noisy signals?
- Speculate about what could you do to the decoder in order to increase the system's tolerance to additive noise.

5. On the front page of your report, please provide an estimate of the average amount of time spent outside of lab by each member of the group.

Laboratory 8

Classification and Vowel Recognition

8.1 Introduction

The ability to recognize and categorize things is fundamental to human cognition. A large part of our ability to understand and deal with the world around us is a result of our ability to classify things. This task, which is generally known as *classification*, is important enough that we often want to design systems that are capable of recognition and categorization. For instance, we want vending machines to be able to recognize the bills inserted into the bill changer. We want internet search engines to classify web pages based on their relevance to our query. We want computers that can recognize and classify speech properly so that we can interact with them naturally. We want medical systems that can classify unusual regions of an x-ray as cancerous or benign. We want high speed digital communication modems that can determine which sequence of, say, 64-ary signals that was transmitted.

There is a vast array of applications for classification. We have actually already seen some of these applications. Detection, which we studied in Labs 1 and 2, is a form of classification where we chose from only two possibilities. In this lab, we consider one popular application of multiple-alternative classification: speech recognition. In particular, we will focus on a simplified version of speech recognition, namely, *vowel classification*. That is, we will experiment with systems that classify a short signal segment of an audio signal which corresponds to a spoken vowel, such as an "ah", an "ee", an "oh", and so on. (We won't deal with how one determines that a given segment corresponds to a vowel.) In the process, we will develop some of the basic ideas behind automatic classification.

One of these basic ideas is that an item to be classified is called an *instance*. For example, if each of 50 short segments of speech must be individually classified, then each segment is considered to be one instance. A second basic idea is that there is a finite set of prespecified *classes* to which instances may belong. The goal of a *classifier system* (or simply a *classifier*) is to determine the class to which a presented instance belongs. A third basic idea is that to simplify the process, the classification of a given instance is based on a set of *feature values*. This set is a relatively small list of numbers that, to an appropriate degree, describes the given instance. For example, a short segment of speech might contain thousands of samples, but we will see that vowel classification can be based on feature sets with as few as two components. A fourth basic idea is that classification is often performed by comparing the feature values for an instance to be classified with sets of feature values that are *representative* of each class. The output of the classifier will be the class whose representative feature values are most similar, in some appropriate sense, to the feature values of the

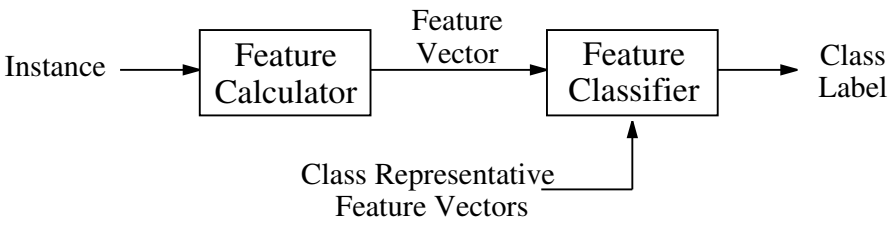


Figure 8.1: Block diagram of a general classifier system.

instance to be classified.

8.1.1 "The Question"

- What is the general framework for performing automatic classification?
- How can we recognize and classify vowels in a speech recognition system?

8.2 Background

8.2.1 An Introduction to Classification

You may recall Lab 7, in which we developed a system for decoding DTMF signals into the sequence of key-presses that produced the original signal. Our DTMF decoder was actually performing classification on each segment of the DTMF signal. Classification is a process in which we examine *instances* of some thing (like an object, a number, or a signal) and try to determine which of a number of groups, or *classes*, each instance belongs to. We can think of this as a labeling process. In our DTMF decoder, for example, we looked at a given segment of the signal and labeled it with a number corresponding to an appropriate key press.

Generally, classification is a two-stage process. Figure 8.1 shows a block diagram of a classifier system. First, we need some information about the instance that we are considering. This information is traditionally referred to as a set of *features*. If we are classifying people, for instance, we might use height, weight, or hair color as features. If we are classifying signals, we might use power, the output of some filter, or the energy in a certain spectral band as features. So that we can deal with our features easily, we generally like to have a set of measurable features to which we can assign numerical *feature values*. When we are using more than one feature to describe an instance, we typically place all of the feature values into a *feature vector*, $\mathbf{f} = (f_1, f_2, \ldots, f_N)$. N is the number of elements in the feature vector and is called the *dimension* of the feature vector. A feature vector is calculated for each instance we wish to classify by measuring the appropriate aspects of that instance. As shown in Figure 8.1, the first block is the "feature calculator," which takes an instance (of a signal, for instance) and produces the set of numerical feature values. For our DTMF decoder, our features were the spectral strength of a given segment of the signal at each DTMF frequency. That is, the feature calculator produced a seven-element feature vector, one for each DTMF frequency.

The second stage of classification, the "feature classifier" (which we have previously called a "decision maker"). uses the feature vectors to decide which class a feature vector belongs to.

Generally, we make this decision by comparing the feature vector for an instance to each member of a set of *representative feature vectors*, one for each class under consideration. The idea is that the feature classifier labels the instance as the class that has the most similar representative feature vector. We will discuss the specifics of the feature classifier after we have presented a classification example.[1]

Before we continue, we should note the relationship between what we previously called "detection" and what we now call "classification". Detection generally refers to binary "signal present" or "signal not present" decisions. For instance in Lab 1, we used energy to decide whether a signal was present or not, and Lab 2 we used correlation to make such decisions. As such, detection, is generally considered to be a special case of the more general notion of classification, which refers to decisions among two or more classes. However, this usage is not universal. For example, "detection" is sometimes used to describe a system that decides which of 64 potential signals was transmitted to a modem, each representing a distinct pattern of 6 bits. This lab assignment also generalizes the idea, used in Labs 1 and 2, that decisions are made on a single number or feature. However, as noted before, Lab 7 also used such a generalization.

8.2.2 A classification example

The easiest way to get a feel for classification problems is to consider an example. Suppose that we have a large number of flowering plants in our garden, each of which belongs to one of two different types, A and B. We know which plant belongs to each type, but they all look very similar. Now, we may know which of our plants belong to which type, but we would also like to be able to classify new plants as either Type A or Type B as we expand our garden. To do this, we will *design* a classifier for these plants. The plants in our garden with known type will form our *design set* (or *training set*) and will be used for designing the classifier.

If we happen to know that Type A plants tend to be taller than Type B plants, this suggests that we might be able to use the plant's height as a feature for classification. Suppose we measure the heights of all of the plants in our garden (our design set) and then plot a histogram of this data. We might see something like Figure 8.2. This is an unusually good case in which we have two readily-

[1] The classifier for the DTMF decoder can be viewed as implicitly operating in this fashion. It is an interesting exercise to find the representative feature vectors implicitly used.

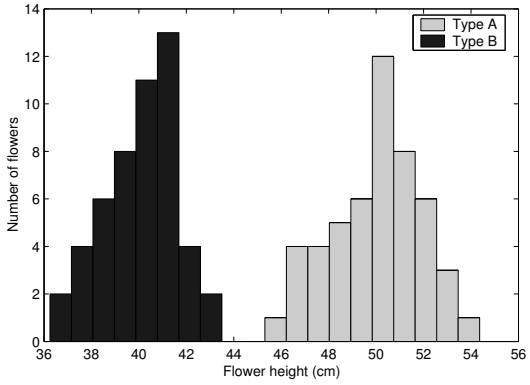


Figure 8.2: A simple example where one feature (plant height) is sufficient to perform classification. This histogram shows how many plants have a given height.

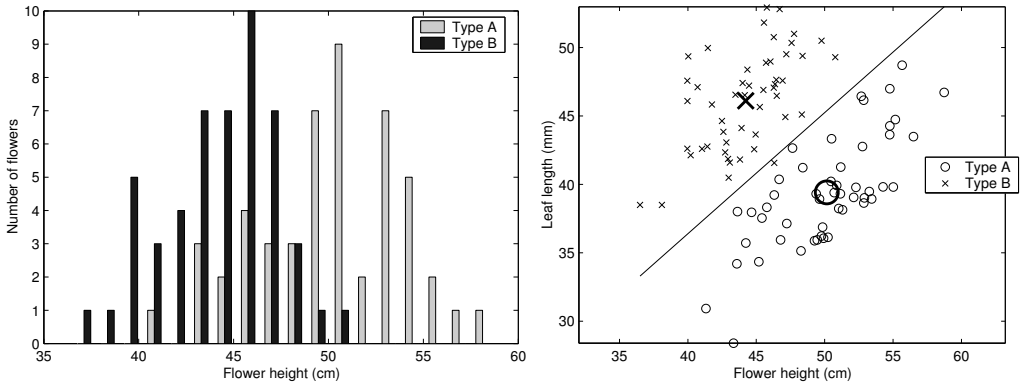


Figure 8.3: An example where a histogram of one feature is *not* sufficient to perform perfect classification (left), but a scatter plot of two features shows a clear separation between the two classes (right).

distinguished *clusters* of feature values. The type A plants form a cluster with heights centered around a mean (i.e. average) of 50 centimeters, and the type B plants form a cluster with heights centered around a mean of 40 centimeters. Most importantly, the two clusters do not overlap. This suggests that classification can indeed be based on plant height.

How do we use this information to classify a new plant (i.e., a new instance)? Intuitively, if the new plant's height is closer to the Type A mean of 50 cm than to the Type B mean of 40 cm, we should classify the plant as type A rather than type B. In this case, we can use a simple threshold test. If a new plant's height is greater than 45 cm (which is halfway between two mean feature values), we classify the new plant as Type A. Conversely, if it's height is less than 45 cm we classify it as Type B. In other words, for each class, we use the mean feature value as the class representative, and we compare the feature value of a new instance to be classified (it's height) to the two means and *decide* the class whose representative feature value (its mean) is closest to the feature value of the given instance.

Figure 8.3 (left) shows a histogram of plant heights in a more troublesome scenario. In this case, Type A plants still tend to be taller than Type B plants, but there are a significant number of plants that we will confuse (that is, misclassify) if we decide exclusively using this one feature. Though we will typically need to deal with some classification error, we can often reduce it by adding more features. Suppose we measure not only the height of the plant but also the average length of its leaves. Now, instead of a histogram, we can look at the training set of features using a *scatter plot*, in which we plot a point for each feature vector in our training set. We put one of the two features along each of the plot's axes. For example, a scatter plot for the two features just mentioned for each plant is shown in Figure 8.3 (right). Here we again see two distinct clusters, which suggests that we can classify with little error by using these two features together.[2]

How do we design a classifier for this case? We cannot simply use a threshold on one of the features. Instead, we will use a more general decision rule, which is based on mean feature vectors and distances between an instance and the mean feature vectors. First, let us consider the two features as a two-dimensional vector $\mathbf{f} = (f_1, f_2)$. Thus, if a plant is 52 cm tall and has leaves with

[2]In this case, classification using either feature by itself will result in many classification errors. That is, by itself, neither feature is sufficient to separate the two clusters. One can see this by *projecting* the scatter plot on to either one of the axes. When we do so, we see that the two feature values of the two classes are intermingled, rather than remaining distinct.

average length of 44 mm, our feature vector is $\mathbf{f} = (52, 44)$. Now, given the feature vectors from each plant in our design set of one type, we want to calculate a *mean feature vector* for plants of that type. Since the mean feature vector indicates the central tendency of each feature in a class, we use it as a *representative* of the entire class. To calculate a mean feature vector in this case, we first take the mean, m_1, of all of the plant heights for plants of one type. Then we take the mean, m_2, of all of the leaf lengths for plants of the same type. The mean feature vector is then $\bar{\mathbf{f}} = (m_1, m_2)$. Note that this is the general procedure for calculating the mean of a set of vectors, regardless of the vector's dimension. On the scatter plot in Figure 8.3, we've plotted the locations of mean feature vectors with large symbols.

As with the one-feature case, we will classify new instances based on how close they are to each of the mean feature vectors. To do this, we still need to know how to calculate distances between two feature vectors. For simplicity, we will calculate distances using the *Euclidean distance measure*[3]. The Euclidean distance between two vectors is simply the straight-line distance between their corresponding points on a scatter plot like that in Figure 8.3. To calculate the distance, d, between two feature vectors (f_1, f_2) and (m_1, m_2), we simply use the formula

$$d = \sqrt{(f_1 - m_1)^2 + (f_2 - m_2)^2} \tag{8.1}$$

Euclidean distance generalizes to any number of dimensions; the general formula can be found later in equation (8.2). Note that the Euclidean distance is essentially the RMS difference (i.e., RMS "error") between two vectors[4], which we have used repeatedly throughout this course. Here, though, we refer to the computation as "Euclidean distance", rather than RMS difference, to motivate a geometric interpretation of classification.

Now that we have designed a classifier for this case, we can finally consider the classification of a new instance. To classify a new instance, we first calculate the distances between that instance's feature vector and the mean feature vectors of each class. Then, we simply classify the instance as a member of the class for which the distance is smallest. Consider what this means in terms of the scatter plot. Given a new instance, we can plot it's feature vector on the scatter plot. Then, we classify based on the nearest mean feature vector. For a two-class case such as that shown in Figure 8.3, there exists some set of points that are equally far from both mean feature vectors. These points form a *decision line* that separates the plane into two halves. We can then classify based on the half of the plane on which a feature vector falls. For example, in Figure 8.3, any plant with a feature vector that falls above the line will be classified as type B. Similarly, any plant with a feature vector that falls below the line will be classified as type A.

With this classification rule, we can correctly classify almost all of our training instances. However, note that we're not classifying perfectly. There is one rogue type B close to the rest of the type A's. In general, though, we will need to accept more error than this.

Of course, two features may not be enough either. If our scatter plot looked like the one in Figure 8.4, then we can still see the two clusters, but we can't perfectly distinguish them based only on these two features. The line we draw for our distance rule will properly classify most of the instances, but many are still classified incorrectly. Once again, we can either accept the errors that will be made or we can try to find another feature to help us better distinguish between the two classes. Unfortunately, visualizing feature spaces with more than two dimensions is rather difficult. However, the intuition we've built for two-dimensional feature spaces extends to higher dimensions. We can calculate mean feature vectors and distances in the roughly the same way regardless of the number of dimensions.

[3]There are a wide variety of possible distance measures; Euclidean distance is certainly not the only choice.

[4]The two calculations actually differ by a scaling factor, since RMS involves a *mean* while Euclidean distance involves a *sum*.

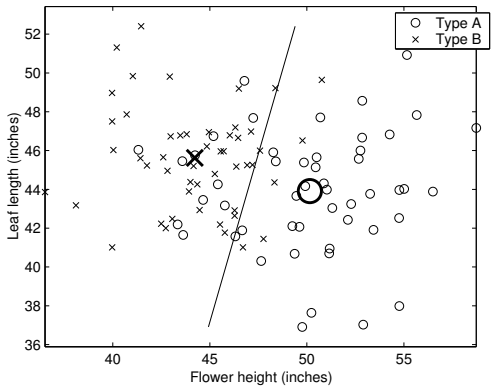


Figure 8.4: An example where two features are not as clearly separated.

8.2.3 A few more classification examples

We've looked at a simple classification task with only two classes, but there are some more examples that are instructive. Consider Figure 8.5(A). In this example, the two clusters fall right on top of one another, so we will have very poor classification performance. This is an example where neither of the features assist classification performance very much. In this case, we need to find better features before we can have much luck with classification. Figure 8.5(B) shows a similar example. Here, feature 2 will help us to improve our classification performance but feature 1 will not. (Can you see why?) Note that it may be worse to have a second feature which is bad than to only have one (good) feature. Unfortunately, determining which features are good and which are bad is nontrivial when we have more than two (or three) features and can no longer visualize the data.

It is also important to realize that we may have more than just two classes in a classification problem. Figure 8.5(C) shows an example in which we have four classes that have distinct clusters in our feature space. The mean feature vectors are indicated on these plots with large markers. Again, we can use the same distance-based decision rule to classify instances. That is, we classify a given instance according to the class whose mean feature vector is closest to its feature vector. We have included (approximate) decision lines on this plot which partition the feature space (i.e., the plane) into four pieces. These indicate which class a given feature vector will be classified as. Of course, multiple classes can be indistinct, too. Figure 8.5(D) shows an example for three indistinct classes. Here, two of the classes ($*$ and $\circ$) are reasonably distinguishable, but we cannot easily separate the third class ($+$) from either of the other two.

8.2.4 Formalizing the feature classifier

In the previous sections, we presented some examples of classifier design and operation. Here, we'll formalize these ideas with respect to the general classifier block diagram shown in Figure 8.1. In particular, we will expand upon the *feature classifier* shown in that block diagram. Figure 8.6 shows an expanded block diagram of the feature classifier[5].

Suppose we need our classifier to decide among C classes and that the classifier will be based on a set of N features, forming a feature vector $\mathbf{f} = (f_1, \ldots, f_N)$. Our feature classifier will rely on

[5]The main goal of any feature classifier is to determine which of a set of representative feature vectors a new instance is most similar to. In this lab, we use Euclidean distance to measure similarity, and so we use a distance-based feature classifier. Other types of feature classifier are also possible, such as a correlation-based feature classifier.

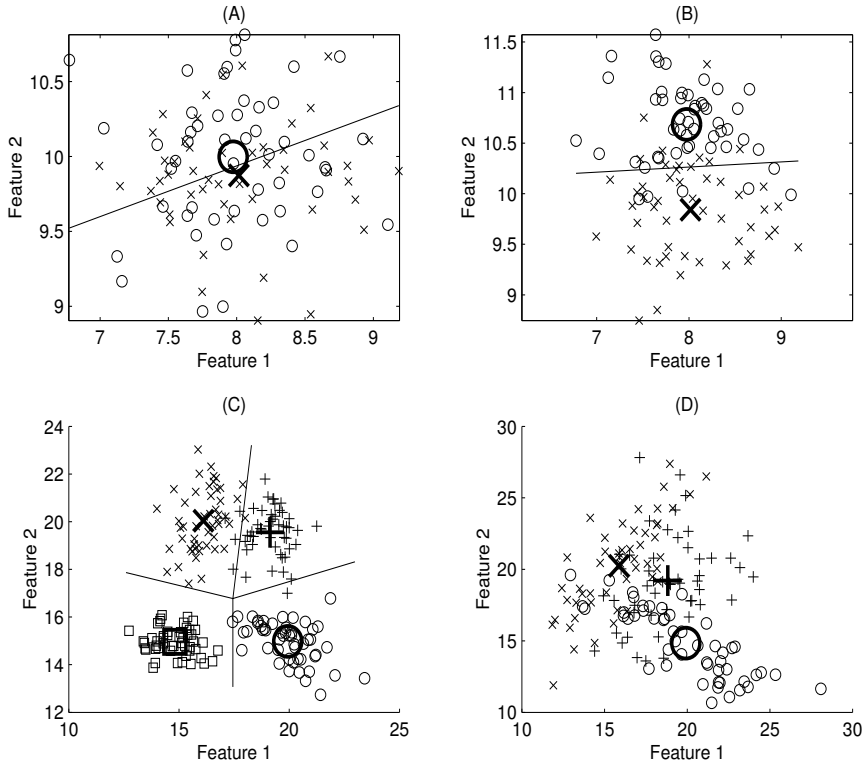


Figure 8.5: (A) Classes overlap, so the features do not allow much discrimination; these are bad features. (B) Feature 2 aids discrimination, but Feature 1 does not. (C) An example with four distinct classes; decision lines are approximate. (D) An example with three indistinct classes.

a set of representative feature vectors, one for each class. We will denote the representative feature vector for the c^{th} class as $\overline{\mathbf{f}}_c = (\bar{f}_{c,1}, \ldots, \bar{f}_{c,N})$, where $c = 1, \ldots, C$.

Given a set of representative feature vectors (the choice of such will be discussed later), we can classify new instances using the feature classifier. The feature classifier (seen in Figure 8.6) has two steps. The first step computes the distances between the input feature vector and each of the class representatives. As we have done in the previous sections, we will use Euclidean distance in our system. Equation (8.1) gives the formula for Euclidean distance in two dimensions. For a general, N-dimensional feature space, we use the following equation. Let $\mathbf{u} = (u_1, \ldots, u_N)$ and $\mathbf{v} = (v_1, \ldots, v_N)$ be two N-dimensional vectors (i.e., arrays with length N). We calculate the Euclidean distance between them as

$$d(\mathbf{u}, \mathbf{v}) = \sqrt{\sum_{i=1}^{N} (v_i - u_i)^2} = \sqrt{(v_1 - u_1)^2 + (v_2 - u_2)^2 + \cdots + (v_N - u_N)^2}. \tag{8.2}$$

Again, we note that, to within a scaling factor, Euclidean distance is equivalent to the root mean squared error between two vectors.

The second step of the feature classifier applies a *decision rule* to select the best class for the input instance. The decision rule that we will use is the *nearest class representative rule*. This simply

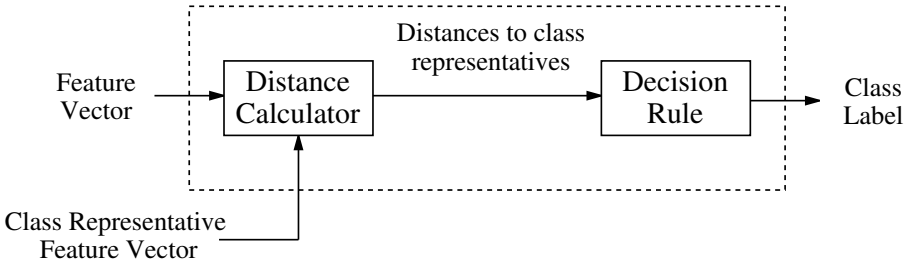


Figure 8.6: Block diagram of a distance-based feature classifier, which makes the decision in a general classifier system.

means that the classifier *decides* the class whose whose representative feature vector is closest (in Euclidean distance) to the feature vector of the instance being classified. That is, if $\mathbf{f}$ is the feature vector for an instance to be classified, then the decision rule decides class c if $d(\mathbf{f}, \bar{\mathbf{f}}_c)$ is less than $d(\mathbf{f}, \bar{\mathbf{f}}_{c'})$ for all other classes[6] c'. Other decision rules, which may weight the distances from the various class representatives, are also possible, but they will not be considered here[7].

Let us now discuss how to choose the class representative feature vectors $\bar{\mathbf{f}}_1, \ldots, \bar{\mathbf{f}}_C$. Finding these vectors is the main aspect in feature classifier design. We have previously suggested that we can find a representative feature vector for a class by taking the mean across some set of instances that belong to that class. We describe this calculation formally as follows. Suppose that we have a set of N-dimensional feature vectors from M instances of a given class c (this the design set of instances for this class). Let $\tilde{\mathbf{f}}_i = (\tilde{f}_{i,1}, \tilde{f}_{i,2}, \ldots, \tilde{f}_{i,N})$ denote the i^{th} such feature vector. We calculate the mean feature vector, $\bar{\mathbf{f}}_c = (\bar{f}_{c,1}, \ldots, \bar{f}_{c,N})$, for this class as

$$\bar{\mathbf{f}}_c = \frac{1}{M} \sum_{i=1}^{M} \tilde{\mathbf{f}}_\mathbf{i} = \frac{1}{M} (\tilde{\mathbf{f}}_\mathbf{1} + \tilde{\mathbf{f}}_\mathbf{2} + \cdots + \tilde{\mathbf{f}}_\mathbf{M}). \tag{8.3}$$

Alternatively, we can say that the j^{th} element of the mean feature vector, $\bar{\mathbf{f}}_c$, is

$$\bar{f}_{c,j} = \frac{1}{M} \sum_{i=1}^{M} \tilde{f}_{i,j} = \frac{1}{M} (\tilde{f}_{1,j} + \tilde{f}_{2,j} + \cdots + \tilde{f}_{M,j}). \tag{8.4}$$

8.2.5 Measuring the performance of a classifier

The performance of a classifier is based on the how many errors it makes. One good way to characterize the performance of a classifier is with a *confusion matrix*, K, which simply measures how often members of one class were confused with members of another class. Specifically, when the classifier recognizes N classes, then the confusion matrix $K = [K_{i,j}]$ is an $N \times N$ matrix, whose element $K_{i,j}$ in the i^{th} row and j^{th} column is the fraction of those times that class j occurs but the

[6] If it should happen that $\mathbf{f}$ is equally closest to two or more class representatives, then an arbitrary choice is made among them.

[7] Note that our DTMF signal classifier from Lab 7 used a simpler "feature classifier" that was based on neither distance nor correlation. However, with a little extra work it could have been formulated as either of these types of classifier, most likely without a degradation of performance.

classifier produces class i. That is,

$$K_{i,j} = \frac{\text{\# of class } j \text{ instances classified as } i}{\text{\# of class } j \text{ instances}} \tag{8.5}$$

For example, the following is confusion matrix for a hypothetical four-class classifier:

$$K = \begin{bmatrix} .9 & .03 & .01 & .02 \\ .03 & .95 & .01 & .03 \\ .05 & .01 & .96 & .1 \\ .02 & .01 & .02 & .85 \end{bmatrix} \tag{8.6}$$

The diagonal elements, $K_{n,n}$, show what fraction of instances from the the n^{th} class were correctly classified. The .9 in the upper left corner, for instance, indicated that 90% of instances from the first class were classified as belonging to the first class. Thus, higher diagonal elements are desirable.

The off-diagonal element $K_{n,m}$ indicates what fraction of instances from the m^{th} class were misclassified as belonging to class n. In the example above, for instance, the .02 in the upper right corner indicates that 2% of instances in the fifth class were incorrectly classified as belonging to the first class. Thus, we hope that off-diagonal elements are as small as possible. The confusion matrix for a perfect classifier will be an identity matrix (i.e., ones on the diagonals, zeros elsewhere).

Data usage when designing classifiers

When designing classifiers and testing their performance, it is important to note that classifiers generally perform better on the training data used in their design than on new data of the same general type. Thus, to objectively assess the performance of a classifier, one must test it on a different data set, usually called a *test set*, than the one on which it was designed. To see why, consider the extreme case in which the training data contains just one feature vector for each class, which becomes the mean feature vector for its class. In this case, the resulting classifier will perfectly classify every feature vector in the training set. However, it may not do very well at all when classifying other data. In more realistic cases where the training data has quite a few instances of each class, the performance of the classifier on the training data will usually be somewhat (but possibly not significantly) better than on test data. Nevertheless, it is widely accepted that testing a classifier on independent data is good practice. Thus, when a certain amount of data is available for design, it is usually divided into two sets — one for training, the other for testing.

To keep things simple, in this lab we will not separate our set of instances into separate design sets and testing sets. Thus, it is important to know that we may not be accurately characterizing our system's performance in the "real world." You will be given an opportunity to test our vowel classifier and see how well it actually performs on your voice. Specifically, all of the vowel instances provided in this lab were taken from a single speaker. How does this affect the performance of the system for *other* speakers? Can you come up with a better set of representative feature vectors (possibly by collecting vowel samples from a variety of speakers)?

8.2.6 Vowel Classification

So far, we have discussed a general-purpose framework for performing classification. In this section, we will specifically discuss how to apply these techniques to the classification of vowels in speech signals.

Vowels in speech are nearly periodic segments of the speech signal. From our studies of the Fourier Series, we know that these segments of the signal are thus approximately equal to the sum

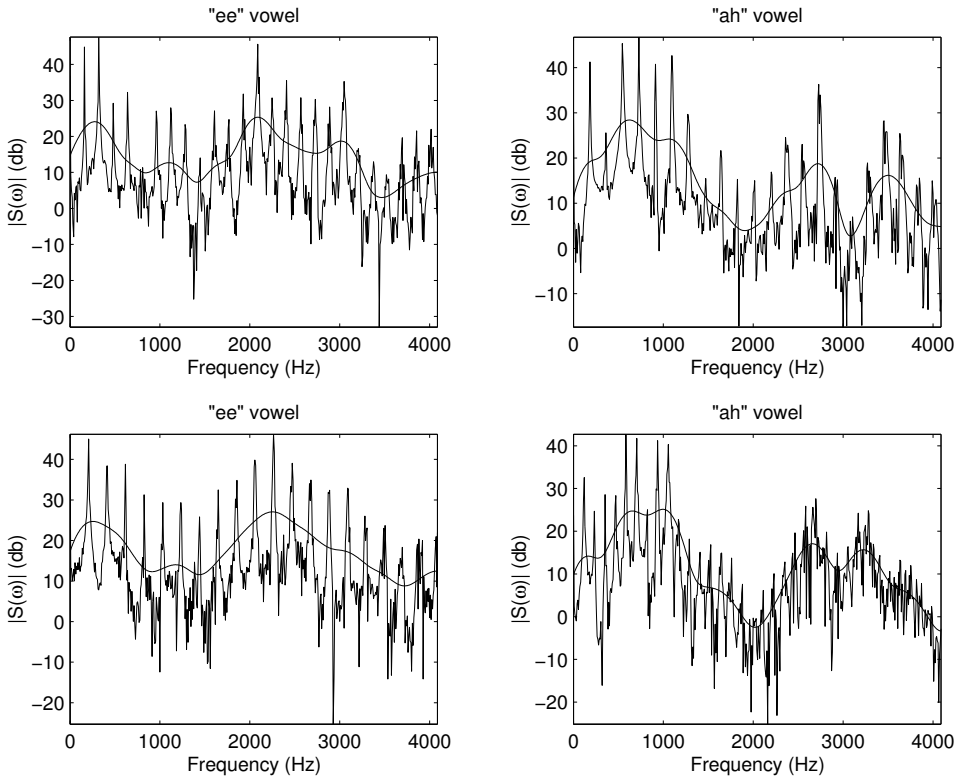


Figure 8.7: The magnitude spectrum (in decibels) of four vowel signals. The plots on the left correspond to two instances of an "ee" vowel, as in the word *tree*. The plots on the right correspond to two instances of an "ah" vowel, as in the word *father*. The solid line is a smoothed version of the spectrum, which shows the general trends of the spectrum.

of sinusoids with harmonically related frequencies. As with the DTMF signals in Lab 7, the time-domain provides relatively little information about the signal. So, as with the DTMF signals, this suggests that we need to examine vowels in the frequency domain. Figure 8.7 shows examples of the magnitude spectrum (in *decibels*[8]) of two different vowels. The plots on the left correspond to an "ee" vowel (as in the word *tree*), while the plots on the right correspond to an "ah" vowel (as in the word *father*). Also shown is a smoothed version of each spectrum, which shows its general trend.

There are a number of interesting things to note about these plots. First, we can see the peaks that correspond to the harmonics that make up the periodic signal. Notice that the peaks are spaced more closely in some plots than others, corresponding to a lower fundamental frequency and thus a longer fundamental period. As illustrated by this figure, though, the fundamental frequency of the signal is independent of the vowel being produced. Notice that the overall shape of the frequency spectrum is different between the two vowels, but remains relatively constant between the two instances of each vowel, as can be seen from the smooth versions. This shape determines the *timbre*[9] of the sound, and, correspondingly, the "sound" of the vowel. Notice that there are peaks in the smoothed

[8]To convert a number, x, into decibels, we use the formula $x_{dB} = 20\log_{10}(x)$.

[9]Pronounced "tambor."

spectrum at various places. These peaks are called *formants*; it is generally known that the position of these formant is the primary feature that distinguishes one vowel from another, i.e. that makes one vowel sound different from another.

Unfortunately, there is no solid definition of a "formant," and they are remarkably difficult to identify automatically. In fact, there is some disagreement as to what constitutes a formant in some cases. In this lab, we'll work with two sets of features that hopefully capture the information contained in the formant positions. In Lab 9, we'll investigate the use of another, somewhat more sophisticated feature for vowel recognition. This feature actually models speech production, and thus should more readily capture the relevant aspects of the vowel signal.

The first feature set that we use in this lab will be the *formant features*. The formant features attempt to locate the formants themselves using a simple algorithm. This algorithm first uses the DFT to compute the spectrum of a short segment of a vowel. Then, the spectrum is smoothed using a weighted averaging filter. Finally, the algorithm returns frequencies of the largest peaks on the smoothed signal that occur above and below 1500 Hz. Thus, there are two formant features, so the resulting feature vector is two-dimensional.

The second feature set, the *filter bank features*, are quite similar to the features used in the DTMF decoder. The filter bank features compute the energy (in decibels) of the speech signal after it has been passed through a bank of six bandpass filters. We will use bandpass filters with center frequencies of 600 Hz, 1200 Hz, 1800 Hz, 2400 Hz, 3000 Hz, and 3600 Hz. Thus, the resulting feature vectors are six-dimensional.

Note that there are a large number of vowels that we could possibly consider. However, for simplicity we will restrict attention to just five vowels: "ee" (as in *tree*), "ah" (as in *father*), "ae" (as in *fate*), "oh" (as in *boat*), and "oo" (as in *moon*). Each of these five vowels will be its own class.

8.3 Some MATLAB commands for this lab

- **Converting a value into decibels:** Expressing a numerical value in *decibels* compresses the range of values using a logarithmic transformation. Thus allows us to see features that might otherwise not be visible. The decibel transformation is particularly useful when looking at the magnitude spectrum of audio signals, since hearing is based on a logarithmic amplitude scale. Given a value `x`, we convert it to decibels using the command

  ```
  >> x_dB = 20*log10(x);
  ```

  This command can be also used to simultaneously convert a vector of values to decibels.

- **Calculating features for a vowel signal:** As indicated, the features we would like to consider in order to classify a vowel signal are based on the signal's spectrum. We provide functions to calculate the two feature sets described in this laboratory. Each function takes an audio waveform, `x`, and (optionally) the sampling frequency in samples per second, `fs`. (If no sampling frequency is specified, a sampling frequency of 8192 samples per second is assumed.) Both functions return a row vector, `y`, that contains the features calculated from the waveform.

  To compute the "formant features," use `calc_formants.m`:

  ```
  >> y = calc_formants(x,fs);
  ```

  Similarly, to compute the "filter bank features," `calc_fbank.m`:


```
>> y = calc_fbank(x,fs);
```

- **Working with features vectors in MATLAB:** In this lab, we will adopt the convention that a *feature vector* is a row vector, and that a set of feature vectors, such as a set of class representatives or a set of testing data, is stored in a matrix such that there is one feature vector per row and one feature per column. This allows us to easily compute mean feature vectors from such a matrix.

  When computing Euclidean distances, note that the computation is almost the same as that which we used for computing RMS error. The only difference is that we replace the *mean* operation by a *summation*.

- **Advanced plotting:** You may recall from Lab 1 that we can use MATLAB's `plot` command to change the color and style of plotted lines. A line-style string consists of as many as three parts. One part specifies a color (for instance, 'k' for black or 'r' for red). Another part specifies the type of markers at each data point (for instance, '`*`' uses asterisks while '`o`' specifies circles). The third part specifies the type of line used to connect the points ('`:`' specifies a dotted line, while '`-`' specifies a solid line). Note that these three parts can occur in any order, and all are optional. If no color is specified, one is chosen automatically. If no marker is specified, a marker will be not be plotted. If a marker is specified but a line type is not, then lines will not be drawn between data points. Thus, the command:

  ```
  >> plot(x1,y1,'rx',x2,y2,'k:');
  ```

  will plot `x1` versus `y1` using red verb-x-'s with no connecting line, and also `x2` with `y2` with a dotted connecting line but no marker. See `help plot` for more details.

  Additionally, we can change the width of lines and the size of markers using additional parameter-pairs. For instance, to increase the line width to 2 and the marker size to 18, use the command

  ```
  >> plot(x1,y1,'rx--','Linewidth',2,'Markersize',18);
  ```

- **Executing the feature classifier:** The function `feature_classifer` is an incomplete function that you will use to automatically classify a feature vector (or a set of feature vectors) based on the distances to a set of representative feature vectors. The function takes two inputs. The first input, `M`, is a matrix of the representative feature vectors, with one feature vector per row. Note that the vector on the first row corresponds to the first class, the second row to the second class, and so on. The second input parameter, `fmatrix`, can either be a single feature vector to be classifed (stored as a *row vector*) or a matrix of feature vectors to be classified, with one feature vector per row. To call `feature_classifier`, use the command:

  ```
  >> labels = feature_classifer(M,fmatrix);
  ```

  The function outputs a *column vector* of class labels, `labels`, with one label for each row of `fmatrix`. The labels are numbers that indicate which representative feature vector the corresponding instance is closest to. Thus, if the first element of `labels` is a 3, it means that the feature vector in the first row of `fmatrix` is closest to the representative feature vector in the third row of `M`.

- **Calculating confusion matrices:** We provide you with a function, `confusion_matrix`, that computes a confusion matrix for you. Note that `confusion_matrix` calls your `feature_classifier` function, so it will not work until you have completed that function. To compute a confusion matrix, we need the matrix of representative feature vectors used by that classifier and a set of testing data for each class. Thus, if our matrix of representative feature vectors is `M` and `class1`, `class2`, and `class3` are matrices that contain feature vectors from our testing set with instances of each of three classes (with one feature vector per row), we compute the 3×3 confusion matrix using the command:

  ```
  >> K = confusion_matrix(M,class1,class2,class3);
  ```

  Note that the function `confusion_matrix` works for any number of classes. The size of the confusion matrix is determined by the number of input parameters.

8.4 Demonstrations in the Lab Section

- Introduction to Classification
- Evaluating performance of a classifier
- Vowel spectra
- Vowel features

8.5 Laboratory Assignment

1. (Examining one vowel instance.) Download the file `lab8_data.mat`. This file contains a variable called `vowel1`, which is a one half-second recording of a vowel sound with a sampling frequency of 8192 samples per second.

   (a) Use `soundsc` to listen to this vowel.
   - To which of the five vowel classes does this vowel belong?

   (b) Take the DFT of `vowel1`. In two subplots of the same figure, plot the magnitude of the DFT and the magnitude of the DFT in decibels. Only plot the *first half* of the DFT coefficients in each plot. Also, make sure that you label the x-axis with the frequency in Hertz, *not* the DFT coefficient number. (Hint: The maximum frequency showing on your plot should be 4096 Hz, which is one half of the sampling frequency.)
   - Include this figure in your report.
   - Compare the two plots. Are there aspects of the spectrum that are easier to see in one of the plots than in the other?
   - From this figure, estimate the fundamental frequency of the vowel sound.
   - Estimate the frequencies (in Hz) of the three most prominent formants.

   (c) Download the files `calc_formants.m` and `calc_fbank.m`. You will use them to calculate features for this vowel.
   - Calculate and include the formant feature vector for this vowel.
   - Compare the calculated features to your estimate of the formant locations.

- Calculate and include the filter bank feature vector for this vowel.
- What are the center frequencies of the filters that have the greatest output amplitude? Compare this to your estimated formant locations.

2. (Mean feature vectors and hand classification.) `lab8_data.mat` also has several other variables, including matrices containing features for 50 instances of each vowel. The variables `ah_form`, `ee_form`, `ae_form`, `oh_form`, and `oo_form` contain formant feature vectors for each vowel. Each matrix has 50 rows (one for each instance) and two columns (one for each feature value). Similarly, the variables `ah_fbank`, `ee_fbank`, `ae_fbank`, `oh_fbank`, and `oo_fbank` contain the filter bank feature vectors with one row per instance.

   (a) (Mean feature vectors) Calculate the mean feature vectors for each vowel class and for both feature classes.
   - Include the five mean formant feature vectors in your report. Make sure you label them.
   - Include the five mean filter bank feature vectors in your report. Again, make sure you label them.

   (b) (Generate a scatter plot) We would like to see how separable the classes are from the formant feature vectors. To do this, you'll create a scatter plot that plots the first formant location versus the second formant location. Plot each of the feature vectors, using a different color and marker symbol for each vowel. Make sure you include a legend. Also, plot the mean vector for each class on your scatter plot. (Hint: To make your mean vectors stand out, you should increase the line width and marker size for just those points.)
   - Include the scatter plot in your report.
   - Interpret this scatter plot. Are all of the classes distinct and easily separated? Do you expect expect any vowels to be frequently confused? Do you expect any vowels to frequently be classified correctly?

   Food for thought: What would happen if we only used one of these two features for classification? Which would give us better classification results? Do you think that a third formant feature might improve class separation?

   (c) (Hand classification) Now, you'll "classify" the signal `vowel1` by hand. To do this, you'll need to compute the distance between the instance and the five mean feature vectors.
   - Compute the distances between the formant feature vector that you generated for `vowel1` and the five mean formant feature vectors.
   - Compute the distances between the filter bank feature vector that you generated for `vowel1` and the five mean filter bank feature vectors.
   - Using the nearest-representative decision rule, use the above results to classify `vowel1`. Do the results for both feature sets agree? If not, which feature set produces the correct answer?

3. (Complete and use the feature classifier code.) In this problem, you'll complete and then use the function, called `feature_classifier.m`, that does this classification for us automatically. As described in the background section, the function takes as input a matrix of representative feature vectors and a matrix of instances to classify. We output a label for each of the instances in the input.

(a) Complete the function. (Hint: You should use two `for` loops. One loops over the rows of the matrix of test instances. Then, for each instance, loop over the rows of `M` and compute the distances. To make the classification for each instance, find the *position* of the smallest distance and store it in `labels`.)

- Include your code in your report.

(b) (Test `feature_classifier` on the formant features.) Place your mean formant feature vectors into a matrix, `M_form` with one feature vector per row. For consistency, put "oo" in the first row, "oh" in the second row, "ah" in the third row, "ae" in the fourth row, and "ee" in the fifth row. Call your `feature_classifier` function using this matrix and `ee_form`. (Hint: To make sure your function works correctly, you should compare its output to the completed and compiled function `feature_classifier_demo.dll`. If you did not successfully complete `feature_classifier`, you can use this demo function throughout the remainder of the lab.)

- What fraction these instances are properly classified?
- Calculate the fraction of the instances that are misclassified as each of the incorrect classes. That is, determine the fraction that are misclassified as "ah," the fraction misclassified as "ae," and so on.
- From this data, what vowel is "ee" most often misclassified as?

(c) Repeat the above with the filter bank features, this time using the matrix `ee_fbank` and generating the matrix `M_fbank`. Use the same order for your classes. (Again, you should compare your function's output to the output of `feature_classifier_demo.dll`).

- What fraction these instances are properly classified?
- Calculate the fraction of the instances that are misclassified as each of the incorrect classes.
- From this data, what vowel is "ee" most often misclassified as?

4. (Compute and interpret confusion matrices.) Download the file `confusion_matrix.m`. In this problem, you will compute and interpret confusion matrices for the two feature classes.

(a) Use `confusion_matrix` to compute the confusion matrix for the formant features. Use `M_form` as your set of class representatives. Use the vowel following vowel order for your remaining input parameters: "oo," "oh," "ah," "ae," and "ee." (This should be the same as the order as the classes in `M_form`).

- Include this confusion matrix in your report. Label each row and column with the corresponding class.
- In Problem 3b, you computed a portion of the confusion matrix. Identify that portion and verify that your results were correct.
- From this confusion matrix, determine how many instances of "ee" vowels were misclassified as "oo" vowels.
- Which vowel is most commonly misclassified using this feature set?

(b) Use `confusion_matrix` to compute the confusion matrix for the filter bank features. Use `M_fbank` as your set of class representatives. Use the vowel following vowel order for your remaining input parameters: "oo," "oh," "ah," "ae," and "ee." (This should be the same as the order as the classes in `M_form`).

- Include this confusion matrix in your report. Label each row and column with the corresponding class.
- In problem 3c, you computed a portion of the confusion matrix. Identify that portion and verify that your results were correct.
- From this confusion matrix, determine how many instances of "ae" vowels were misclassified as "ah" vowels.
- Which vowel is most commonly misclassified using this feature set?

(c) Finally, compare the two confusion matrices.

- Which feature set has the best performance overall?
- Based on the performance of these classifiers, comment on the spectral similarities between the various vowels. That is, are any of the vowel classes particularly like any of the other vowel classes?

5. On the front page of your report, please provide an estimate of the average amount of time spent outside of lab by each member of the group.

Food for thought:

- *As was mentioned in the background section, all of the vowel instances used here were taken from a single speaker. As such, this classifier will probably perform better on that speaker's vowels than on the rest of the population. The compiled functions* `formant_classifier.dll` *and* `fbank_classifier.dll` *let you test the classifier designed here on your own voice. When you execute either of these functions (which require no input parameters),* MATLAB *will immediately record one quarter-second from the microphone, compute the features, and classify the vowel. Use this function to test the classifier on your own voice for various vowels. Record how well it does on each vowel. Is performance better or worse than what is indicated by the confusion matrices you calculated?*
- *The above function also returns the set of features that it computed from the recorded vowel. If you collect these features into series of matrices of testing data, you can use* `confusion_matrix.m` *to formally compute the performance of this classifier on your voice.*
- *How well does the classifier work on* other *people's voices? Are there certain people for whom it works very well? Very poorly? Does it work if we vary the pitch?*
- *The compiled functions listed above take an optional input parameter, which is a matrix of representative feature vectors. Since the current classifier is designed using only one speaker's vowels, maybe you can improve the performance by coming up with a better set of representative feature vectors. To do this, consider gathering a set of vowels from a number of different speakers and combining them into a set of mean feature vectors. Can you improve the performance of the system?*

Laboratory 9

Filter Design, Modeling, and the z-Plane

9.1 Introduction

So far, we've been considering filters as systems that we design and then apply to signals to achieve a desired affect. However, filtering is also something that occurs everywhere, without the intervention of a human filter designer. At sunset, the light of the sun is filtered by the atmosphere, often yielding a spectacular array of colors. A concert hall filters the sound of an orchestra before it reaches your ear, coloring the sound and adding pleasing effects like reverberation. Even our own head, shoulders, and ears form a pair of filters that allows us to localize sounds in space.

Quite often, we may wish to recreate these filtering effects so that we can study them or apply them in different situations. One way to do this is to *model* these "natural" filters using simple discrete-time filters. That is, if we can measure the response of a particular system, we would often like to design a filter that has the same (or a similar) response.

One of the goals for this laboratory is to introduce the use of discrete-time filters as models of real-world filters. In particular, we will examine how to apply a modeling approach to understanding vowel signals. This in turn will suggest a way that we might improve the performance of the vowel classifier we developed in Lab 8 using an automatic modeling method.

Another goal of this lab is to present a method of filter design called *pole-zero placement design*. Working with this method of filter design is extremely useful for building an intuition of how the z-plane "works" with respect to the frequency domain that you are already familiar with. The design interface that we use for this task should help you to develop a graphical understanding of how poles and zeros affect the frequency response of a system. We will use this design methodology both to design a traditional "goal-oriented" lowpass filter, and to do some filter modeling.

9.1.1 "The Question"

- How can we *design* filters for certain purposes?
- How can we model vowel production using discrete-time filters?

9.2 Background

9.2.1 Filters and the z-transform

Previously, we have presented the general time-domain input-output relationship for a causal filter[1] given by the convolution sum:

$$y[n] = x[n] * h[n] = \sum_k h[k]x[n-k] = \sum_k x[k]h[n-k] \ , \tag{9.1}$$

where $x[n]$ is the input signal, $y[n]$ is the output signal, and $h[n]$ is the filter impulse response. Using the z-transform techniques described in Chapter 7 of *DSP First*, we can also describe the input/output relationship in the z-domain as

$$Y(z) = H(z)X(z) \ , \tag{9.2}$$

where $X(z)$ is the z-transform of $x[n]$, which is the complex-valued function, defined on the complex plane[2] by

$$X(z) = \sum_n x[n]z^{-n} \ , \tag{9.3}$$

where $Y(z)$ is the z-transform of $y[n]$, defined in a similar fashion, and where $H(z)$ is the *system function* of the filter, which is a complex-valued function defined on the complex plane by one of the following equivalent definitions:

1. The system function is the z-transform of the filter impulse response $h[n]$, i.e

$$H(z) = \sum_n h[n]z^{-n} \ . \tag{9.4}$$

2. For $X(z)$ and $Y(z)$ as defined above, the system function is given by

$$H(z) = \frac{Y(z)}{X(z)} \ . \tag{9.5}$$

The system function has a very important relationship to the frequency response of a system, $\mathcal{H}(\hat{\omega})$. The system function evaluated at $e^{j\hat{\omega}}$ is equal to the frequency response evaluated at frequency $\hat{\omega}$. That is,

$$\mathcal{H}(\hat{\omega}) = H(e^{j\hat{\omega}}) \ . \tag{9.6}$$

We can derive this result from equation 9.4. If we let $z = e^{j\hat{\omega}}$, then we know that $H(z) = H(e^{j\hat{\omega}}) = \sum_n h[n]e^{-j\hat{\omega}n}$. This is simply the definition of a system's frequency given its impulse response $h[n]$.

[1] Note that in this lab, we will only be concerned with causal filters.

[2] The *complex plane* is simply the set of all complex numbers. The real part of the complex number is indicated by the x-axis, while the imaginary part is indicated by the y-axis.

9.2.2 FIR Filters and the z-transform

For a causal FIR filter, one can easily determine the system function using either of the equivalent definitions given above. However, let us highlight the use of the second definition, which will be useful in the next subsection where the first definition is difficult to apply. In particular, for a causal FIR filter with coefficients $\{b_0, \ldots, b_M\}$, the general time-domain input-output relationship for a causal FIR filter is given by the difference equation

$$y[n] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] + \cdots + b_M x[n-M] \; . \tag{9.7}$$

Taking the z-transform of both sides of this difference equation yields

$$\begin{aligned} Y(z) &= b_0 X(z) + b_1 X(z) z^{-1} + b_2 X(z) z^{-2} + \cdots + b_M X(z) z^{-M} \\ &= X(z)(b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3} + \cdots + b_M z^{-M}) \; , \end{aligned} \tag{9.8}$$

where we have used the fact that the z-transform of $x[n-n_o]$ is $X(z)z^{-n_o}$. Dividing both sides of the above by $X(z)$ gives the system function:

$$H(z) = \frac{Y(z)}{X(z)} = b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3} + \cdots + b_M z^{-M}. \tag{9.9}$$

Notice that $H(z)$ is a polynomial of order M. We can factor the above complex-valued polynomial as[3]

$$H(z) = K(1 - r_1 z^{-1})(1 - r_2 z^{-1})(1 - r_3 z^{-1}) \cdots (1 - r_M z^{-1}) \; , \tag{9.10}$$

where K is a real number called the *gain*, and $\{r_1, \ldots, r_M\}$ are the M *roots* or *zeros* of the polynomial, i.e. the values r such that $H(r) = 0$. We typically assume that the filter coefficients b_k are real. In this case, the zeros may be real or complex, and if one is complex, then its complex conjugate is also a zero. That is, complex roots come in conjugate pairs.

The very important point to observe now from equation (9.10) is that the system function $H(z)$ of a causal FIR filter is completely determined by its gain and its zeros. Therefore, we can think of $\{K, r_1, \ldots, r_M\}$ as one more way to describe a filter[4]. We will see that when it comes to designing an FIR filter to have a certain desired frequency response, the description of the filter in terms of its gain and its zeros is by far the most useful. In other words, the best way to design a filter to have a desired frequency response (e.g., a low pass filter) is to appropriately choose its gain and zeros. One may then find the system function by multiplying out the terms of equation (9.10), and then picking off the filter coefficients from the system function. For example, the number multiplying z^{-3} in the system function is the filter coefficient b_3. The specific procedure will be described shortly.

The fact that we may design the frequency response of a causal FIR filter by choosing its zeros[5] stems from the following principle:

> If a filter has a zero r located on the unit circle, i.e. $|r| = 1$, then $\mathcal{H}(\angle r) = 0$, i.e. the frequency response has a *null* at frequency $\angle r$. Similarly, if a filter has a zero r located close to the unit circle, i.e. $|r| \approx 1$, then $\mathcal{H}(\angle r) \approx 0$, i.e. the frequency response has a *dip* at frequency $\angle r$. In either case, $\mathcal{H}(\hat{\omega}) \approx 0$, when $\hat{\omega} \approx \angle r$.

[3] The Fundamental Theorem of Algebra guarantees that $H(z)$ factors in this way.

[4] Previous ways of describing a filter have included the filter coefficients, the impulse response sequence, the frequency response function, and the system function.

[5] The gain does not affect the shape of the frequency response.

The above fact follows from the property that if $\hat{\omega} = \angle r$ and $|r| = 1$, then $e^{j\hat{\omega}} = r$, and so

$$\mathcal{H}(\hat{\omega}) = H(e^{j\hat{\omega}}) = H(r) = 0\ . \tag{9.11}$$

A similar statement shows $\mathcal{H}(\hat{\omega}) \approx 0$ when $|r| \approx 1$ and/or $\hat{\omega} \approx \angle r$.

From this fact, we see that we see that we can make a filter block a particular frequency, i.e. create a null or a dip in the frequency response, simply by placing a zero on or near the unit circle at an angle equal to the desired frequency[6]. On the other hand, the frequency response at frequencies corresponding to angles that are not close to these zeros will have large magnitude. The filter will "pass" these frequencies. The specific procedure to design such a filter is the following.

1. Choose frequencies $\hat{\omega}_1, ..., \hat{\omega}_L$ at which the frequency response should contain a null or a dip.
2. Choose zeros $r_i = \rho_i e^{j\hat{\omega}_i}$, $i = 1, \ldots, L$, with $\rho_i = 1$ or $\rho_i \approx 1$, depending upon whether a null or a dip is desired at frequency $\hat{\omega}$. For each $\hat{\omega}_i \neq 0$ choose also a zero r_j that is the complex conjugate of r_i. Let M be the total number of zeros chosen.
3. Form the system function $H(z) = K(1 - r_1 z^{-1}) \times \cdots \times (1 - r_M z^{-1})$, where K is a gain that we also choose.
4. Cross multiply the factors of $H(z)$ found in the previous step so as to express $H(z)$ as a polynomial whose terms are powers of z^{-1}.
5. Identify the FIR filter coefficients $\{b_0, \ldots, b_M\}$, which are simply the coefficients of the polynomial found in the previous step, as shown in equation (9.9).

9.2.3 IIR filters and rational system functions

We now consider IIR filters. The general time-domain input-output relationship for a causal IIR filter is given by the difference equation

$$\begin{aligned} y[n] \quad = b_0 x[n] \quad & + b_1 x[n-1] + b_2 x[n-2] + \cdots + b_M x[n-M] \\ & + a_1 y[n-1] + a_2 y[n-2] + \cdots + a_N y[n-N]\ . \end{aligned} \tag{9.12}$$

Here, we have the usual FIR filter coefficients, b_k, but we also have another set of coefficients a_k, which multiply *past values of the filter's output*. We will call the b_k's the *feedforward coefficients* and the a_k's the *feedback coefficients*[7]. If the a_k's are zero, then this filter reduces to a causal FIR filter.

As an example, consider the simple IIR filter with difference equation:

$$y[n] = x[n] + \frac{1}{2} y[n-1] \tag{9.13}$$

What is the impulse response of this filter? If we assume[8] that $y[n] = 0$ for $n < 0$, one can straightforwardly show that the impulse response is

$$h[n] = \left(\frac{1}{2}\right)^n, \quad n \geq 0, \tag{9.14}$$

[6] Since non-real zeros must occur in conjugate pairs, we must also place a conjugate zero on the unit circle, i.e. a zero whose angle is the negative of the first.

[7] In some texts (and in MATLAB), the feedback coefficients are defined as the negatives of the a_k coefficients given here. Because of this, you should always be sure to check which convention is used.

[8] This assumption is one of the "initial rest conditions" discussed in chapter 8 of *DSP First*.

which is never zero for any positive n. (Note that the impulse response is generally not so simple to compute; this is an unusual case where the impulse response can be obtained by inspection.) Thus, by introducing feedback terms into our difference equation, we have produced a filter with an infinite impulse response, i.e., an IIR filter.

In general, computing the system function by taking the z-transform of the resulting infinite impulse may not be trivial because of the required infinite sum, and also because it may be difficult to find the impulse response. However, we can use the fact that $H(z) = Y(z)/X(z)$ to determine the system function. To do this, we first collect the $y[n]$ terms on the left side of the equation and take the z-transform of the result.

$$y[n] - a_1 y[n-1] - \cdots - a_N y[n-N] = b_0 x[n] + b_1 x[n-1] + \cdots + b_M x[n-M] \tag{9.15}$$

$$Y(z) - a_1 Y(z) z^{-1} - \cdots - a_N Y(z) z^{-N} = b_0 X(z) + b_1 X(z) z^{-1} + \cdots + b_M X(z) z^{-M} \tag{9.16}$$

$$Y(z)(1 - a_1 z^{-1} - \cdots - a_N z^{-N}) = X(z)(b_0 + b_1 z^{-1} + \cdots + b_M z^{-M}) \tag{9.17}$$

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + \cdots + b_M z^{-M}}{1 - a_1 z^{-1} - \cdots - a_N z^{-N}} \tag{9.18}$$

Equation (9.18) shows the general form of the system function of an IIR filters. Since it is the ratio of two polynomials, it is called a *rational function*[9].

Just as we could factor the polynomial in equation (9.9), we can do the same with equation (9.18) to yield

$$H(z) = K \frac{(1 - r_1 z^{-1})(1 - r_2 z^{-1})(1 - r_3 z^{-1}) \cdots (1 - r_M z^{-1})}{(1 - p_1 z^{-1})(1 - p_2 z^{-2})(1 - p_3 z^{-1}) \cdots (1 - p_N z^{-1})}. \tag{9.19}$$

The roots of the polynomial in the numerator, $\{r_1, \ldots, r_M\}$, are again called the *zeros* of the system function. The roots of the polynomial in the denominator, $\{p_1, \ldots, p_N\}$ are called the *poles* of the system function. K is again a gain factor that determines the overall amplitude of the system's output. As before, the zeros are complex values where $H(z)$ goes to zero. The poles, on the other hand, are complex values where the denominator goes to zero and thus the system function goes to infinity[10]. Again, we typically assume that the filter coefficients b_k and a_k are real, so both the poles and zeros of the system function must be either purely real or must appear in complex conjugate pairs.

Just as we could completely characterize an FIR filter by its gain and its zeros, we can completely characterize an IIR filter by its gain, its zeros, and its poles. As in the FIR case, this is typically the most useful characterization when designing IIR filters. As before, if the system function has zeros near the unit circle, then the filter magnitude frequency response will be small at frequencies near the angles of these zeros. On the other hand, if there are poles near the unit circle, then the magnitude frequency response will be be large at frequencies near the angles of these poles. With FIR filters we could directly design filters to have nulls or dips at desired frequencies. Now, with IIR filters, we can design peaks in the frequency response, as well as nulls. The specific procedure is the following.

1. Choose frequencies $\hat{\omega}_1, ..., \hat{\omega}_L$ at which the frequency response should contain a null, a dip, or a peak.

[9] This is a generalization of the terminology that the ratio of two integers is called a *rational number*.

[10] Technically, because of a division by zero, $H(z)$ is undefined at the location of a pole. However, the magnitude of the system function becomes very large in the neighborhood of a pole.

2. Choose zeros $r_i = \rho_i e^{j\hat{\omega}_i}$ at those frequencies at which a null or a dip should occur, with $\rho_i = 1$ or $\rho_i \approx 1$, as desired. For each such $\hat{\omega}_i \neq 0$, choose also a zero r_j that is the complex conjugate of r_i. Let M be the total number of zeros chosen.

3. Choose poles $p_i = \rho_i e^{j\hat{\omega}_i}$ at those frequencies at which a peak should occur, with $\rho_i = 1$ or $\rho_i \approx 1$ as desired. For each such $\hat{\omega}_i \neq 0$ choose also a pole p_j that is the complex conjugate of p_i. Let N be the total number of poles chosen.

4. Form the system function $H(z) = K\frac{(1-r_1 z^{-1})\times\cdots\times(1-r_M z^{-1})}{(1-p_1 z^{-1})\times\cdots\times(1-p_N z^{-1})}$, where K is a gain that we also choose.

5. Cross multiply the factors of $H(z)$ found in the previous step and express $H(z)$ as the ratio of two polynomials whose terms are powers of z^{-1}.

6. Identify the IIR filter coefficients $\{a_0, \ldots, a_N, b_0, \ldots, b_M\}$, which are simply the coefficients of the polynomials found in the previous step, as shown in equation (9.18).

Poles and zeros at the origin and at infinity

Here, we have defined our system functions in terms of negative powers of z. This is because our general forms for FIR and IIR filters are defined in terms of *time delays*, and multiplication of the z-transform of some signal $X(z)$ by z^{-1} is equivalent to a time delay of one sample. However, there are may be "hidden" poles and zeros when we express a system function in this matter.

Consider first the system function for our FIR filter given by (9.9). If we try to evaluate this system function at $z = 0$, we will immediately find that we are dividing by zero. Thus, there is actually a pole at the origin of this system function. To reveal such "hidden" poles and zeros, we express the system function in terms of positive powers of z. To do so, we multiply by $\frac{z^M}{z^M}$, which yields

$$H(z) = \frac{b_0 z^M + b_1 z^{M-1} + b_2 z^{M-2} + b_3 z^{M-3} + \cdots + b_M}{z^M}. \tag{9.20}$$

By the Fundamental Theorem of Algebra, we know that the numerator polynomial has M roots, and thus the system has M zeros. However, the denominator, z^M, has M roots as well, all at $z = 0$. This means that our causal FIR system function has M poles at the origin.

In some cases, like the previous example, we find extra poles at the origin. In other cases, we find extra zeros at the origin. For example, the filter $y[n] = y[n-1] + x[n]$, has $H(z) = \frac{1}{1-z^{-1}} = \frac{z}{z-1}$, from which we see there is one zero at the origin. In still other cases we find zeros at infinity[11]. For example, the filter $y[n] = x[n-1]$, has $H(z) = z^{-1} = \frac{1}{z}$, from which we see that there is a zero at infinity. In still other cases, we find combinations of the previous cases. We will call poles and zeros located at the origin or at infinity*trivial poles and zeros* because they do not affect the system's magnitude frequency response[12]. In this laboratory, we will primarily be concerned with *nontrivial poles and zeros* (those not at the origin or at infinity).

Note that there will always be the same number of poles and zeros in a linear time-invariant system, including both trivial and nontrivial poles and zeros. The total number equals the M or N, whichever is larger. Such facts are useful for checking to make sure that you have accounted for all poles and zeros in a system.

[11] A zero at infinity means that $|H(z)| \to 0$ as $|z| \to \infty$. It can be shown that a causal filter can never have a pole at infinity.

[12] Trivial poles and zeros *do* affect the phase (and thus the delay or time shift) of a system.

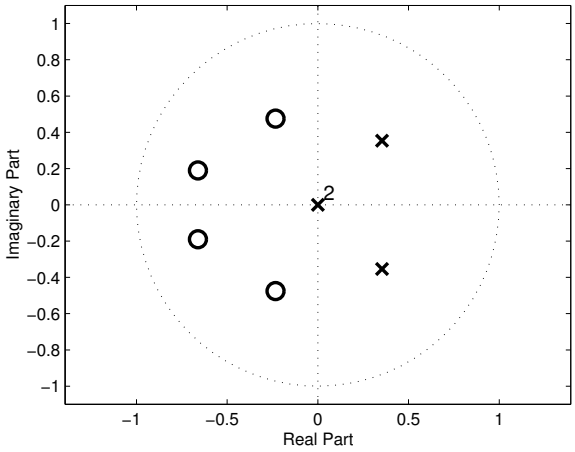


Figure 9.1: A pole-zero plot of an IIR filter.

Note also that if one chooses filter coefficients such that the numerator and denominator contain an identical factor, i.e. if $r_i = p_j$ for some i and j, then these factors "cancel" each other, i.e. the filter is equivalent to a filter whose system function has neither factor.

Pole-zero plots

It is often very useful to graphically display the locations of a system's poles and zeros. The standard method for this is the *pole-zero plot*. Figure 9.1 shows an example of a pole-zero plot. This is a two-dimensional plot of the z-plane that shows the unit circle, the real and imaginary axes, and the position of the system's poles and zeros. Zeros are typically marked with an 'o', while poles are indicated with an 'x'. Sometimes, a location has multiple poles and zeros. In this case, a number is marked next to that location to indicate how many poles or zeros exist there. Figure 9.1, for instance, shows four zeros (two conjugate pairs), two "trivial" poles at the origin, and one other conjugate pair of poles. Recall that zeros and poles near the unit circle can be expected to have a strong influence on the magnitude frequency response of the filter.

9.2.4 Graphical interpretation of the system function

If we take the magnitude of $H(z)$, we can think of $|H(z)|$ as defining a (strictly positive) *surface* over the z-plane for which the *height* of the surface is given as a function of the complex number z. Figure 9.2 shows an example of just such a surface. This system function has two zeros (which form a complex conjugate pair) and two poles at the origin. Notice that the unit circle is outlined on the surface $|H(z)|$. The height of the surface at $z = e^{j\hat{\omega}}$ (i.e., on the unit circle) defines the magnitude of the frequency response, $|\mathcal{H}(\hat{\omega})|$, which is shown to the right of the surface.

On Figure 9.2, we can see two points where the surface $|H(z)|$ goes to zero; these are the zeros of the system function. Notice how the surface is "pulled down" in the vicinity of these zeros, as though it has been "tacked to the ground" at the location of the zeros. Near the system's zeros, the magnitude frequency response has a low point because of the influence of the nearby zero. Also notice how the surface is "pushed up" at points far from the zeros; this is another common characteristic of system function zeros. (Since the two poles in this figure are at the origin, they have no effect on the

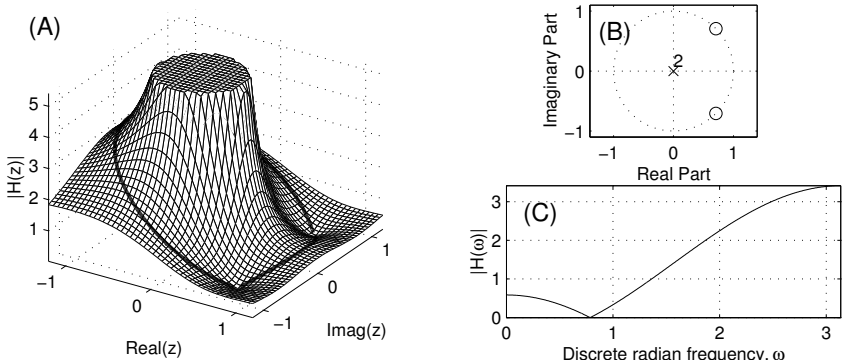


Figure 9.2: (A) The z-plane surface defined by the system function $H(z) = (1 - e^{j\pi/4}z^{-1})(1 - e^{-j\pi/4}z^{-1})$. (B) The corresponding pole-zero plot. (C) The corresponding magnitude frequency response.

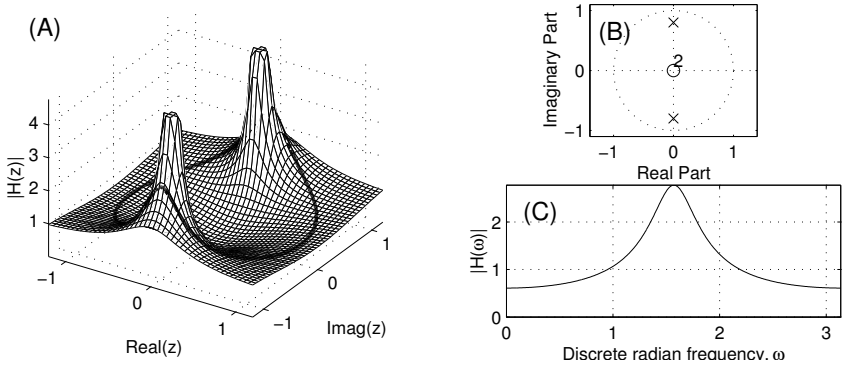


Figure 9.3: (A) The z-plane surface defined by the system function $H(z) = \frac{1}{(1-0.8e^{j\pi/2}z^{-1})(1-0.8e^{-j\pi/2}z^{-1})}$. (B) The corresponding pole-zero plot. (C) The corresponding magnitude frequency response.

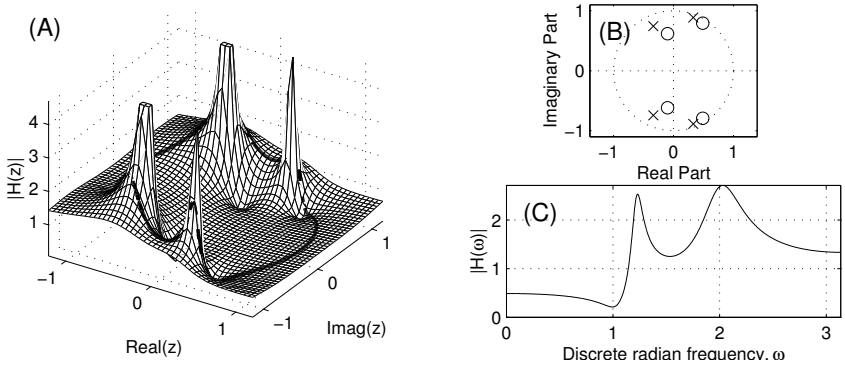


Figure 9.4: (A) The z-plane surface for a complicated system function with four poles and four zeros. (B) The corresponding pole-zero plot. (C) The corresponding magnitude frequency response.

system's magnitude frequency response.) Thus, the magnitude frequency response has higher gain at points far away from the zeros.

Figure 9.3 shows the surface $|H(z)|$ as defined by a different system function. This system function has two poles (which form a complex conjugate pair) and two zeros at the origin. Notice how the poles "push up" the surface near them, like poles under a tent. The surface then typically "drapes" down away from the poles, getting lower at points further from them. The magnitude frequency response here has a point of high gain in the vicinity of the poles. (Again, the zeros in this system function are located at the origin, and thus do not affect the magnitude frequency response.)

Figure 9.4 shows the surface for a system function which has poles and zeros interacting on the surface. This system function has four poles and four zeros. Notice the tendency of the poles and zeros to cancel the effects of one another. If a pole and a zero coincide exactly, they will completely cancel. If, however, a pole and a zero are very near one another but do not have exactly the same position, the z-plane surface must decrease in height from infinity to zero quite rapidly. This behavior allows the design of filters with rapid transitions between high gain and low gain.

9.2.5 Poles and stability

System poles cause the system function to go to infinity at certain values of z because we are dividing by zero. On the one hand, this can have the desirable effect of raising the magnitude frequency response at certain frequencies. On the the other hand, this can have some undesirable side effects. One somewhat significant problem is introduced if we have a pole outside the unit circle. Consider the following filter, for instance:

$$y[n] = x[n] + 2y[n-1] \tag{9.21}$$

This filter has a single pole at $z = 2$. What is this system's impulse response? If the input is $x[n] = \delta[n]$ and $y[n] = 0$ for $n < 0$, then at $n = 0$, $y[n] = 1$. Then, every $y[n]$ after that is equal to twice the value of $y[n-1]$. The value of this impulse response *grows* as time goes on! This system is *unstable*[13]. Unstable filters cause severe problems, and so we wish to avoid them at all costs. As a general rule of thumb, you can keep your filters from being unstable by keeping their poles strictly inside the unit circle. Note that the system's zeros do not need to be inside the unit to maintain stability.

9.2.6 Filter design using manual pole-zero placement

In this laboratory, we will explore a method of filter design in which we place poles and zeros on the z-plane in order to match some target frequency response. You will be using a MATLAB graphical user interface to do this. The interface allows you to place, delete, and move poles and zeros around the z-plane. The frequency response will be displayed in another figure and will change dynamically as you move poles and zeros. To keep the filter's coefficients real, you will design by placing a pair of poles and zeros on the z-plane simultaneously.

The approach for this method depends somewhat on the type of filter that we wish to design. If we want an FIR filter (i.e., a filter that has no poles), we need to use zeros to "pin down" the frequency response where it is low, and allow the frequency response to be pushed upwards in regions where there are no zeros. Note that if we put a zero right on the unit circle, we introduce null in the frequency response at that point. Conversely, the closer to the origin that we place a zero, the less effect it will have on the frequency response (since it will begin to affect all points on the unit

[13]Technically, it is *bounded input, bounded output* (BIBO) unstable because the input has a limited magnitude but the output does not.

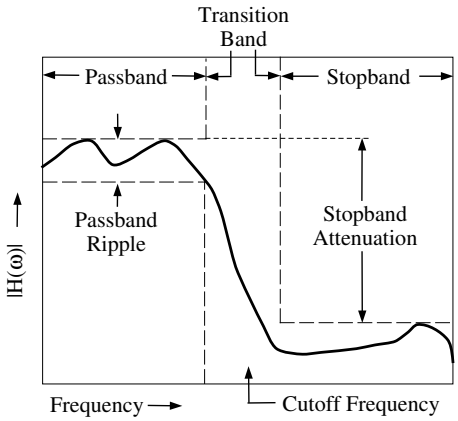


Figure 9.5: An illustration of the various bands of a lowpass filter.

circle roughly equally). You might use the example of the running average filter and bandpass filters (given in Chapter 7 of *DSP First*) as a prototype of how to use zeros to design FIR filters using zero placement.

If we wish to design an IIR filter (with both poles and zeros), it usually makes sense to start with the poles since they typically affect the frequency response to a greater extent. If the frequency response that we are trying to match has peaks on it, this suggests that we should place a pole somewhere near that peak (inside the unit circle). Then, use zeros to try to pull down the frequency response where it is too high. As with zeros, poles near the origin have relatively little effect on the system's filter response.

Regardless of which type of filter we are designing, there are a couple of methodological points that should be mentioned. First, moving a pole or zero affects the frequency response of the entire system. This means that we cannot simply optimize the position of each pole-pair and zero-pair individually and expect to have a system which is optimized overall. Instead, after adjusting the position of any pole-pair or zero-pair, we generally need to move many of the remaining pairs to compensate for the changes. This means that filter design using manual pole-zero placement is fundamentally an iterative design process.

Additionally, it is important that you consider the filter's gain. Often we cannot adjust the overall magnitude of the frequency response using just poles and zeros. Thus, to match the frequency response properly, you may need to adjust the filter's gain up or down. The pole-zero design interface that you will use in this Lab includes an edit box where you can change the gain parameter. Alternately, by dragging the frequency response curve, you can change the gain graphically. A related idea is that of *spectral slope*. By having a pair of poles or zeros inside the unit circle and near the real axis, we can adjust the overall "tilt" of the frequency response. As we move the pair to the right and left on the z-plane, we can adjust the slope of the system's frequency response up and down.

Note that there are automatic filter design methods which do not require manual placement of poles and zeros. In Section 9.2.8 we discuss one such method.

9.2.7 Design of Standard Filter Types

Many of the filters that we wish to design and use belong to one of four standard types: lowpass, highpass, bandpass, or bandstop. These filters are characterized by a *passband* (a band of frequencies which are relatively unaltered by the filter) and a *stopband* (a band of frequencies which are significantly *attenuated*, or decreased in amplitude, by the filter). The locations of the passband and stopband are what characterize the different filter types. For instance, a lowpass filter has a passband which contains low frequencies and a stopband which contains high frequencies, while a bandpass filter has a single passband that is surrounded by two stopband regions. Between the passband and the stopband is a *transition band* in which the filter's frequency response changes from high to low. The location of the transition band in frequency determines the filter's *cutoff frequency*. In this lab, we will specify the cutoff frequency by using the two frequencies that bound the transition band.

When designing these types of filter, there are a number of different design goals that we may attempt to achieve. For instance, we may wish to have a very flat frequency response in the *passband* of the filter. Unfortunately, it can be difficult to achieve a flat frequency response over some frequency region. Instead, the frequency response usually varies somewhat over that region; this variation is called *ripple*. Thus, one of design criteria may be to minimize *passband ripple*. In this lab, we define the passband ripple as the (positive) decibel value of the ratio between the maximum and minimum filter gains in the passband.

Another common goal is to try to minimize the gain in the stopband of the filter relative to the gain in the passband. That is, we wish to maximize the *stopband attenuation*. In this lab, we define the stopband attenuation as the decibel ratio between the maximum filter gain in the passband and the maximum filter gain in the stopband.

Figure 9.5 shows the passband, transition band, and stopband for a lowpass filter. The figure also illustrates the the passband ripple and stopband attenuation. In Problem 2 of this assignment, you will design a lowpass filter to maximize stopband attenuation.

9.2.8 Modeling Vowel Production

In Lab 8, we discussed some of the properties of vowel production in speech, but we did not examine the mechanisms behind vowel production. In order to gain a better understanding of vowel production and how we can model it using discrete-time filters, we need to introduce some theory.

Speech production is primarily governed by the *larynx* (or voice box) and the *vocal tract*. Figure 9.6 shows a diagram of the larynx and vocal tract. When we speak a vowel, the lungs push a stream of air through the larynx and the *vocal folds* (commonly referred to as the *vocal chords*). Given the appropriate muscular tension, this stream of air causes the vocal folds to vibrate[14]. This in turn creates a nearly periodic fluctuation in air pressure passing through the larynx. The fundamental frequency of vocal fold vibration is typically around 100 Hz for males and 200 Hz for females.

This fluctuating air stream then passes through the vocal tract, which is the airway leading from the larynx and through the mouth to the lips. The positions of the tongue, lips, and jaw serve to shape the vocal tract, with different positions creating different vowel sounds. The different sounds are produced as the vocal tract shapes the spectrum of the pressure signal coming from the larynx. Depending upon the vocal tract configuration, different frequencies of the spectrum are emphasized; from Lab 8, we know these frequencies as *formants*. When whispering a vowel, the lungs push air through the larynx, but the vocal folds do not vibrate. In this case, the air pressure fluctuation is quite noise-like and generally is not periodic. Nevertheless, the tongue, lips and jaw shape the vocal tract just as before to make the various vowel sounds.

[14] In fact, during normal production the vocal folds open and close completely on each cycle of the vibration.

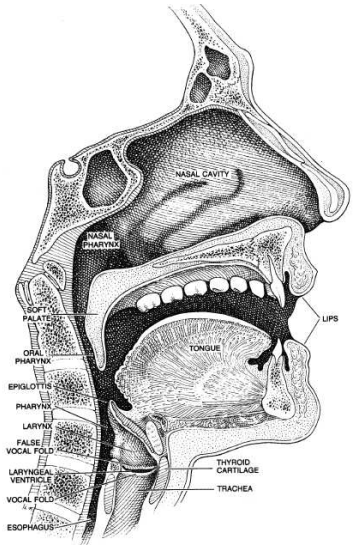


Figure 9.6: A diagram showing the larynx and vocal tract.

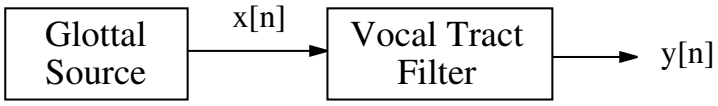


Figure 9.7: A block diagram of the source-filter model of speech production.

Note that the above description is only accurate for vowels and so-called *voiced consonants* like "m" and "n." Most consonant sounds are produced using the tongue, lips, and teeth rather than the vocal cords. We will not consider consonants in this lab.

It is traditional to model speech production using a *source-filter model*. Figure 9.7 shows a block diagram of the source-filter model. The first block is the *glottal source*, which takes as input a fundamental frequency and produces a periodic signal (the *glottal source signal*) with the given fundamental frequency. The signal produced is typically modeled as a periodic pulse train. To a first approximation, we can assume that spectrum of this pulse train is composed of equal amplitude harmonics. The glottal source signal is meant to be analogous to the signal formed by the air pressure fluctuations produced by the vibrating vocal cords. Note that to model whispering, the glottal source signal can be modeled using random noise rather than a pulse train.

The second block of the source-filter model is the vocal tract filter. This is a discrete-time filter that mimics the spectrum-shaping properties of the vocal tract. Since we are assuming a source signal with equal-amplitude harmonics, the vocal tract filter provides the spectral envelope for our output signal. That is, when we filter the source signal with fundamental frequency $\hat{\omega}_0$ radians per sample, the k^{th} harmonic of the output signal will have an amplitude equal to the filter's magnitude frequency response evaluated at $k\hat{\omega}_0$. This is illustrated in Figure 9.8 which shows a particular example. The magnitude spectrum of the glottal source signal is shown on top, the magnitude frequency response of the vocal tract filter is shown in the center, and the magnitude spectrum of the output signal, which is the signal that models the specific vowel signal. One may clearly see that, as

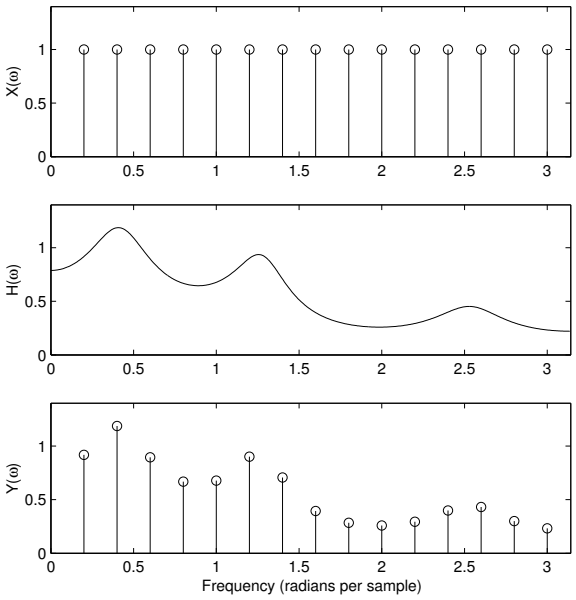


Figure 9.8: A plot of the magnitude spectrum of a glottal source signal, the frequency response of a vocal tract filter, and the magnitude spectrum of the output signal.

desired, the envelope of the spectrum of the vowel signal model matches the spectrum of the vocal tract filter.

In Problem 3 of this assignment, we will make such source-filter models for particular vowel signals, by measuring the spectrum of the vowel signal and designing an IIR vocal tract filter whose frequency response approximates this spectrum.

Typically, our vocal tract filter can have relatively few filter coefficients (i.e., approximately 10-20 coefficients). Further, the acoustics of the vocal tract suggest that this filter should be IIR. Often, the vocal tract is modeled using an *all-pole filter* which has no nontrivial zeros. This is because an acoustic passageway like the vocal tract primarily affects a sound through *resonances*. A resonance is a part of a system that tends to vibrate at a certain *resonant frequency*, thus amplifying that frequency in signals passed through them. The feedback form of an IIR filter is a direct implementation of resonance; this is how IIR filters are able to produce high gain at certain frequencies. Using this simple model of speech production, it is possible to synthesize artificial vowels.

All-pole analysis and vowel classification

In the last lab, we explored some features for vowel classification that were based on two measures of spectral energy in a vowel signal. The development of the source-filter model, however, suggests an acoustically motivated feature for vowel classification. If we assume that the vocal tract can be modeled with a low-order discrete-time filter, then the vocal tract filter captures all of the relevant information about which vowel has been produced. Variations such as fundamental frequency and type of vowel production (i.e., voiced or whispered) are restricted to the glottal source and can be neglected. Using samples of the frequency response of the vocal tract filter as features for vowel classification has been shown to produce good classification results.

Fortunately, there are nice mathematical tools for deriving all-pole filter models automatically

from a time-domain waveform. These tools, fit the spectrum of a time-domain signal with poles in a least-squares sense. Note that these tools work directly with the time-domain waveform rather than its spectrum; typically, they return the resulting a_k feedback coefficients for a filter with those poles, rather than the locations of the poles themselves. We will explore these tools for all-pole analysis in the laboratory assignment, and we will compare classification performance using features based on these models to the performance we achieved with our other feature sets from Lab 8.

9.3 Some MATLAB commands for this lab

- **Calculating the frequency response of IIR Filters:** Previously we have used `freqz` to compute the frequency response of FIR filters. We can use the same command to compute the frequency response of an IIR filter. If our filter is defined by feedforward coefficients b_k stored in a vector `B` and feedback coefficients a_k stored in a vector `A`, we compute the frequency response at 256 points using the command:

  ```
  >> [H,w] = freqz(B,A,256);
  ```

  As with FIR filters, MATLAB's convention for the b_k coefficients is `B(1)` $= b_0$, `B(2)` $= b_1$, ..., `B(M+1)` $= b_M$. MATLAB's convention for the a_k coefficients is `A(1)` $= 1$, `A(2)` $= -a_1$, ..., `A(N+1)` $= -a_N$. Both b_k and a_k are given as defined in equation 9.18. `H` contains the frequency response and `w` contains the corresponding discrete-time frequencies. Alternatively, we can compute the frequency response only at a desired set of frequencies. For example, the command

  ```
  >> [H,w] = freqz(B,A,[pi/4, pi/2, 3*pi/4]);
  ```

  returns the frequency response of the filter at the frequencies $\pi/4$, $\pi/2$, and $3\pi/4$.

- ***Pole-Zero Place 3-D*:** In this laboratory, we will primarily be exploring filter design using manual pole-zero placement. To help us do this, we will be using a MATLAB graphical user interface (GUI) called *Pole-Zero Place 3-D*. *Pole-Zero Place 3-D* allows you to place, move, and delete poles and zeros on the z-plane, and provides immediate feedback by displaying the filter's frequency response and the $|H(z)|$ surface. Additionally, it calculates some useful statistics for assessing the quality of a particular filter design.

  To run this program you need to download two different files: `pole_zero_place3d.m` and `pole_zero_place3d.fig`. To begin *Pole-Zero Place 3-D*, simply execute[15] the command

  ```
  >> pole_zero_place3d;
  ```

  Once the program starts, the GUI window shown in Figure 9.9 will appear. The axis in the upper left of the window shows a portion of the z-plane with the unit circle. In the lower left is an axis that displays the frequency response of the system. In the lower right is a 3-D axis which displays a 3-D graph of the $|H(z)|$ surface[16].

  The interface allows you to do a wide variety of things.

[15] This program was designed to run using Windows systems running MATLAB 6 or higher; it will not work with previous versions of MATLAB. It should work with Unix operating systems running MATLAB 6, but this has not been tested.

[16] This surface plot requires significant computation, and thus it can be toggled on and off using the *View 3-D* checkbox in the upper right.

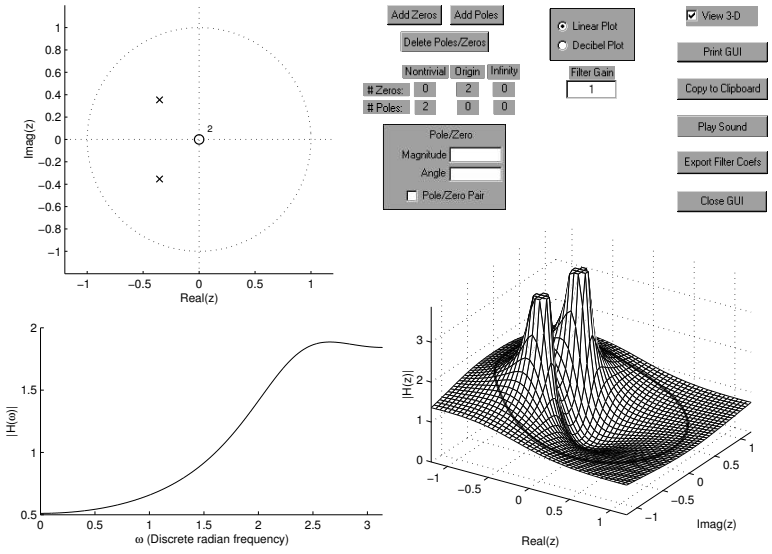


Figure 9.9: The GUI window for *Pole-Zero Place 3-D*.

1. To add a poles or zeros to the z-plane, click the *Add Zeros* or *Add Poles* button and then click on the z-plane plot in the upper left of the GUI. The state of the *Place pair* checkbox determines whether a single (real) pole or zero is added, or whether a conjugate pair is added.

   Note that the program also adds the hidden poles and zeros that accompany nontrivial poles and zeros. Specifically, for each zero that is added a pole is added at the origin, or if there is already at least one zero at the origin, instead of adding a pole at the origin, one zero at the origin is removed, i.e. cancelled. Moreover, for each pole that is added, a zero is added at the origin, or if there are already poles at the origin, one pole is removed, i.e. cancelled. The system does not allow one to place zeros at infinity, and it can be shown that zeros at infinity will not be induced by any other choices of poles or zeros.

2. To move a real pole or zero (or a conjugate pair of complex poles or zeros), you must first select the pole/zero by clicking on one member of the pair. Then, you can drag it around the z-plane, use the arrow keys to move it, or move it to a particular location by inputting the magnitude and angle (in radians) in the *Magnitude* and *Angle* edit boxes.

3. To delete a pole or zero (or pair), select it and hit the *Delete Poles/Zeros* button. Again, the system will maintain an equal number of poles and zeros by also removing poles or zeros from the origin as necessary. This may also have the effect of no longer cancelling other poles and zeros, and thus the total number of poles and zeros that appear at the origin will change.

4. To change the filter's gain, you can either use the *Filter Gain* edit box or you can click-and-drag the blue frequency response curve in the lower left.

5. To toggle between linear amplitude and decibel displays in the lower two plots, select the desired radio button above the *Filter Gain* edit box.

6. To rotate the 3-D $|H(z)|$ plot, simply click-and-drag the axes in the lower right of the GUI. To enable or disable the 3-D plot, toggle the *View 3-D* checkbox in the upper right

of the GUI

7. To begin with an initial filter configuration defined by the feedforward coefficients, `B`, and the feedback coefficients, `A`, start the program with the command

   ```
   >> pole_zero_place3d(B,A);
   ```

   This is useful if you wish to start continue working on a design that you had previously saved. You may set either of these parameters to empty (`[]`) if you do not wish to specify the filter coefficients.
8. To print the GUI window, you can either use the *Copy to Clipboard* button to copy an image of the figure into the clipboard[17], or you can print the figure using the *Print GUI* button.
9. To save your current design, use the *Export Filter Coefs* button. The feedforward and feedback coefficients will be stored in the variables `B_pz` and `A_pz`, respectively.
10. To hear your filter's response to periodic signal with equal-amplitude harmonics, press the *Play Sound* button. This is particularly useful when using the GUI to design vocal tract filters for vowel synthesis.

- ***Pole-Zero Place 3-D* – Filter Matching Mode:** In "filter matching mode," you specify samples of a desired transfer function at harmonically related frequencies and try to match that transfer function. The GUI plots a red curve or stem plot along with the frequency response function; this is the response we wish to match. Two edit boxes labeled *Linear Matching Error* and *Decibel Matching Error* indicate how closely your filter matches the desired frequency response. The matching error values are computed as the RMS error between the desired frequency response and your filter design in both linear amplitude and in decibels.

  To start the GUI in this mode, use the following command:

  ```
  >> pole_zero_place3d(B,A,filter_gains,fund_frq);
  ```

  `filter_gains` are the values of the desired filter frequency response at harmonically related frequencies, and `fund_frq` is the fundamental frequency (in radians per sample) of the harmonic series at which `filter_gains` are defined.

- ***Pole-Zero Place 3-D* – Lowpass Design Mode:** In "lowpass design mode," you specify the maximum frequency of the passband and the minimum frequency of the stopband (both in radians per sample). To start the GUI in this mode, use the following command:

  ```
  >> pole_zero_place3d(B,A,[pass_max,stop_min]);
  ```

  In this mode, the GUI computes the passband ripple and the stopband attenuation of your lowpass filter design. You can use these measures to evaluate your filter design. The figure in the lower right also displays the passband and stopband of the filter, with appropriate minima and maxima.

- **Converting between filter coefficients and zeros-poles:** Given a set of filter coefficients, we often need to determine the set of poles and zeros defined by those coefficients. Similarly, we often need to take a set of poles and zeros and compute the corresponding filter coefficients. There are two MATLAB commands that help us do this. First, if we have our filter coefficients stored in the vectors `B` and `A`, we compute the poles and zeros using the commands

[17] Windows operating systems only


```
>> zeros = roots(B);
>> poles = roots(A);
```

This is because the system zeros are simply the roots of the numerator polynomial whose coefficients are the numbers in `B`, while the system zeros are simply the roots of the denominator polynomial whose coefficients are the numbers in `A`. We can see this in equation 9.18. To convert back, use the commands

```
>> B = poly(zeros);
>> A = poly(poles);
```

Note that we loose the filter's gain coefficient, K, with both of these conversions.

- **Generating pole-zero plots:** Frequently, we'd like to use MATLAB to make a pole-zero plot for a filter. If our filter is defined by feedforward coefficients `B` and feedback coefficients `A` (both row vectors), we can generate a pole-zero plot using the command:

  ```
  >> zplane(B,A);
  ```

  Alternately, if we have a list of poles, `p`, and a list of zeros, `z`, (both column vectors) we can use the following command:

  ```
  >> zplane(z,p);
  ```

  An example of a pole-zero plot resulting from this command is shown in Figure 9.1.

- **Automatic all-pole modeling:** Using the MATLAB command `aryule`, we can compute an all-pole filter model for a discrete-time signal. That is `aryule` automatically finds an all-pole filter whose magnitude frequency response that in some sense matches the magnitude frequency response of the signal. If signal is given by `signal`, the command

  ```
  >> A = aryule(signal,N);
  ```

  returns the filter feedback coefficients a_k as a vector `A`. The parameter N indicates how many poles we wish to use in our filter model. Once we have `A`, we can compute the filter's frequency response at 256 points using `freqz` as

  ```
  >> [H,w] = freqz(1,A,256);
  ```

9.4 Demonstrations in the Lab Section

- The z-transform, system functions, and IIR filters
- The system function "surface," $|H(z)|$
- Using *Pole-Zero Place 3-D* for filter design
- Vocal tract modeling

9.5 Laboratory Assignment

1. (Fit an FIR filter's frequency response.) Download the files `pole_zero_place3d.m`, `pole_zero_place3d.fig`, and `lab9_data.mat`. In this problem, you will get familiar with the *Pole-Zero Place 3-D* program for filter design using pole-zero placement.

   In `lab9_data`, the variable `FIR_fr` contains samples of the frequency response for a simple FIR filter with six zeros. Execute *Pole-Zero Place 3-D* using the command

   ```
   >> pole_zero_place3d([],[],FIR_fr,2*pi/8192);
   ```

   Use the GUI to find an FIR filter with six nontrivial zeros that matches the frequency response of the original filter. You should be able to get the linear matching error to be less than 0.1. (Hint: The original filter had all six of its zeros inside the unit circle, so yours should as well.)

   - Include the GUI window with your matching filter in your report. In this and the following problems, make sure that it is possible to read the filter evaluation scores on your printout. (Note: The easiest way to include the GUI window is to use the *Copy to Clipboard* button on a Windows machine. After hitting the button, wait for the GUI to flash white and then paste the result into your report.)
   - What are the filter coefficients b_k and a_k for your filter?
   - Where are the zeros on the z-plane? Give your answers in rectangular form.

2. (Design a lowpass filter.) In this problem, we will use the "lowpass design mode" of *Pole-Zero Place 3-D* to design some lowpass filters, as described in Section 9.2.7. For the various parts of this problem, use the command

   ```
   >> pole_zero_place3d([],[],2*pi*[1500 2000]/8192);
   ```

   This sets the filter transition band to 1500 Hz to 2000 Hz if we assume a sampling rate of 8192 samples per second.

   (a) (Design an FIR lowpass filter to maximize stopband attenuation.) First, let's see what we can do with just zeros (that is, with FIR filters). Using only six nontrivial zeros (i.e., three zero pairs), design a lowpass filter with a stopband attenuation of at least 30 dB. (Remember, we want our stopband attenuation to be as large as possible). For now, take note of your filter's passband ripple, but don't worry about minimizing it.

   - Include the GUI window with your matching filter in your report.
   - What are the filter coefficients b_k and a_k for your filter?
   - Where are the zeros on the z-plane? Give your answers in rectangular form.

   Food for thought: Using just zeros, try to find a way to minimize the passband ripple. What does this do to your stopband attenuation? Try this with more zeros, but don't use any poles.

   (b) (Design an IIR lowpass filter to maximize stopband attenuation.) There are two primary benefits to the use of IIR filters. First, it is very easy to get very high gain at certain frequencies. This lets us design a lowpass filter with very high stopband attenuation. Using a single pair of nontrivial poles, design a lowpass filter that has a stopband attenuation greater than 60 dB. Use the same transition band as in the previous problem. Again, you should take note of the passband ripple, but don't worry about minimizing it.

- Include the GUI window with your matching filter in your report.
- What are the filter coefficients b_k and a_k for your filter?
- Where are the poles on the z-plane? Give your answers in rectangular form.

(c) (Design an IIR lowpass filter for both high stobpand attenuation and low ripple.) The second benefit of IIR filters is the ability to achieve fast transitions between high gain and low gain. Among other things, this allows us to transition between the passband and stopband more quickly, which in turn allows us to achieve relatively high stopband attenuation with low passband ripple.

Once again using the same transition band, design a lowpass filter with a passband ripple of less than 2 dB and a stopband attenuation of at least 20dB. You may use as many poles and zeros as you wish, but it is possible to meet these criteria with only two poles and four zeros. (Hint: use decibel mode to help you increase the stopband attenuation, and linear mode to help you decrease the passband ripple.)

- Include the GUI window with your matching filter in your report.
- What are the filter coefficients b_k and a_k for your filter?
- Where are the poles and zeros on the z-plane? Give your answers in rectangular form.

Food for thought: For a more interesting challenge, design a lowpass filter with a passband ripple of less than 1 dB and a stopband attenuation of 60 dB. This can be done with six poles and six zeros, but you might want to use more than this.

3. (Matching a vowel signal's spectrum using pole-zero placement.) `lab9_data.mat` contains the variable `vowel2`, which is a short vowel signal sampled at 8192 Hz. In this problem we will find a filter model of the vocal tract used to produce this vowel.

   (a) First, let's examine the signal itself.

   - Use `soundsc` to listen to `vowel2`. What vowel does this signal represent?
   - As a measure of the spectrum, plot the magnitude DFT coefficients for this signal in decibels versus frequency in Hertz. Use only the first half of the DFT coefficients, where the maximum value on the x-axis is one half of the sampling rate.
   - From this plot, estimate the fundamental frequency of the vowel signal in Hertz.

   (b) (Design a vocal tract filter with both poles and zeros.) `lab9_data.mat` contains the variables `vowel2_amps`, which contains the amplitudes of the harmonics that make up `vowel2`. We'll use the amplitudes to find a vocal tract filter for this vowel. Start *Pole-Zero Place 3-D* using the command

   ```
   >> pole_zero_place3d([],[],vowel2_amps,2*pi*frq/8192);
   ```

   where `frq` is the fundamental frequency (in Hz) that you estimated. (Hint: The red stems should go all the way across the frequency response plot. Also, there should be one stem for each element of `vowel2_amps`. If not, you have probably estimated your fundamental frequency incorrectly.)

   Set the GUI to "decibel plot" and find a filter with six nontrivial poles and six nontrivial zeros that makes the decibel matching error as small as possible. You should be able to get the decibel error below 4.2. (It is possible to achieve a decibel error of 3.65.)

   - Include the GUI window with your matching filter in your report.

- What are the filter coefficients b_k and a_k for your filter?
- Where are the poles and zeros on the z-plane? Give your answers in rectangular form.

(c) (Design a vocal tract filter with only poles and zeros.) Now, repeat the above for a filter with 10 poles nontrivial and no nontrivial zeros (except for those at the origin). You should be able to achieve a decibel matching error below 2.6. (It is possible to achieve a decibel error of 2.1.)

- Include the GUI window with your matching filter in your report.
- What are the filter coefficients b_k and a_k for your filter?
- Where are the poles on the z-plane? Give your answers in rectangular form.

If you are working on a computer with audio capability, you should use the *Play Sound* button to listen to a synthesis of the vowel signal. Note that the all-pole model produces less error with fewer total coefficients. This suggests that all-pole filters are more appropriate for vocal tract modeling.

4. (All-pole modeling and classification) In this problem, we'll look at an automatically generated all-pole model of `vowel2`, and we'll see how we can use this model to help us improve our vowel classifier performance from Lab 8.

(a) (Automatically generate an all-pole filter model.) Use `aryule` to compute an all-pole model for `vowel2` with 10 poles.

- What are the resulting feedback coefficients?
- Make a pole-zero plot of the resulting filter.
- Use `freqz` to plot the frequency response of this filter and the frequency response of the filter you found in Problem 3c in two subplots of the same figure. Display the two frequency responses in decibels. (Note: The two frequency responses in decibels may be offset by some constant, which corresponds to a scaling of the original spectrum.)

(b) (Construct and evaluate the vowel classifier.) Here, we'll consider 16 samples of the frequency response of an all-pole filter to be a potential feature vector for vowel classification. `lab9_data.mat` contains five matrices which contain all-pole feature vectors for the same vowel instances we examined in Lab 8. They are `oo_ap`, `oh_ap`, `ah_ap`, `ae_ap`, and `ee_ap`.

- As you did in Lab 8, compute the mean all-pole feature vectors for each of the five vowel classes. Make sure you label which mean vector belongs with which class.
- Combine the above vectors into a matrix of mean feature vectors. Then, use `confusion_matrix.m` (from Lab 8) to calculate the confusion matrix for the all-pole features. As you did in Lab 8, use the vowel order "oo," "oh," "ah," "ae," and "ee". (Note: if you were unsuccessful at completing `confusion_matrix` in Lab 8, you can use the compiled function `confusion_matrix_demo.dll` for this problem.)
- Compare this confusion matrix with the two confusion matrices that you computed in Lab 8. Is the performance of the classifier better with this feature class? Note any similarities between the three confusion matrices.

5. On the front page of your report, please provide an estimate of the average amount of time spent outside of lab by each member of the group.

Commonly Used MATLAB commands

Elementary Math Functions (help elmat)

abs	atan	exp	log	rem	sqrt
acos	ceil	fix	log10	round	tan
angle	conj	floor	mod	sign	
asin	cos	imag	real	sin	

Graphing and Plotting Functions (help plot)

axis	figure	line	print	stem	xlabel
bar	grid	loglog	semilogx	subplot	ylabel
clf	hold	plot	semilogx	text	zoom
close	legend	polar	shg	title	

Relational and Logical Functions (help ops)

all	eq (==)	ge (>=)	isempty	isnan	not (~)
and (&)	exist	gt (>)	isfinite	le (<=)	or (|)
any	find	ischar	isinf	lt (<)	strcmp

Working With Variables (help general, help elmat)

who	length	clear	exist	nan	zeros
whos	size	end	isinf	inf	ones

General Purpose Functions (help general)

exit	quit	help	helpdesk	which	lookfor

Programming and Control Flow (help lang)

break	disp	end	if	pause	try
case	else	error	input	return	warning
catch	elseif	for	otherwise	switch	while

File and Directory Functions (help general, help iofun)

cd	fclose	load	path	save	what
dir	fopen	mkdir	rmdir	type	

Text Input/Output (help iofun, help strfun)

input	keyboard	sprintf	disp	return	num2str

Debugging Commands (help debug)

dbclear	dbquit	dbstatus	dbstop	dbup
dbcont	dbstack	dbstep	dbtype	keyboard

Special Symbols (help ops, help punct)

+	-	*	.*	/	./
^	.^	,	;	:	...
'	=	nan	inf	%	

10 Useful MATLAB Facts

1. MATLAB starts indexing it's arrays from 1 rather than from 0.
2. Use the up-arrow to recall previous commands. If you type in a few characters and then hit the up-arrow, MATLAB will try to find a previous command that started with those characters.
3. When indexing matrices, the indices are always given `a(row,column)`. Similarly, `size(a)` returns a two-element vector `[num_rows, num_columns]`.
4. Semicolons at the end of a line are not necessary; they simply suppress output.
5. If I multiply (or divide, or exponentiate) two arrays without using the dot-operators, I probably won't get what I'm expecting (unless I want to do matrix multiplication).
6. We concatenate arrays (and strings) using square brackets. To do so *horizontally*, we separate the arrays with spaces or commas:

   ```
   >> [ones(3), zeros(3)])
   ```

   To do so *vertically*, we separate the arrays with a semicolon:

   ```
   >> [ones(3); zeros(3)])
   ```

7. When a function returns multiple parameters, we use square brackets to retrieve them:

   ```
   >> [max_value, index] = max([4.3, 2.9, 8.6, 6.3, 1.0])
   ```

   Otherwise, only one parameter is returned.

8. Most MATLAB commands (like `min`, `max`, `sum`, `prod`, and a host of others) work on matrices by operating down each column individually. Thus, after executing this command:

   ```
   >> [max_value, index] = max(eye(6))
   ```

   `max_value` has a vector of six ones (since the maximum value in each column is 1) and `index` is a vector containing the row number of the 1 in each column.

9. The `end` keyword is exceptionally useful when indexing into arrays of unknown size. Thus, if I want to return all elements in a vector but the first and last one, I can use the command:

   ```
   >> x(2:end-1)
   ```

   which is equivalent to the command:

   ```
   >> x(2:length(x)-1)
   ```

10. MATLAB automatically resizes arrays for you. Thus, if I want to add an element on to the end of a vector, I can use the command:

    ```
    >> x(end+1) = 5;
    ```

