University of Michigan
EECS 206 Laboratory Manual

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

January 3, 2003

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 patient and helpful comments during the development and revision of the laboratories.

MAB, DLN, GHW
August 2002

Minor corrections by JF
Jan. 2003

Contents

Preface **iii**

Contents **v**

List of Figures **xii**

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 . . . 2

3 Using MATLAB: The basics . . . 2
- 3.1 Starting MATLAB . . . 2
- 3.2 How to get help . . . 3
- 3.3 Using MATLAB as a calculator (with variables) . . . 3

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

5 Array Arithmetic . . . 6

6 Indexing . . . 7
- 6.1 Basic indexing . . . 7
- 6.2 Single number indexing . . . 7
- 6.3 Vector indexing . . . 7
- 6.4 Finding the size of an array . . . 8
- 6.5 Vector indexing to modify arrays . . . 8
- 6.6 Conditional statements and the “find” command . . . 8

7 Data Visualization . . . 9
- 7.1 Using “plot” . . . 9
- 7.2 Interpolation; line and point styles . . . 9
- 7.3 Axis labels and titles . . . 10
- 7.4 Commands related to “plot” . . . 10
- 7.5 Plotting with an x-axis . . . 10

7.6 Plotting multiple vectors on the same figure . . . 10
7.7 Legends . . . 11
7.8 Putting several axes on one figure . . . 11
7.9 Two-dimensional arrays . . . 11
8 Programming in MATLAB . . . 12
8.1 Paths and working directories . . . 12
8.2 Types of command files in MATLAB . . . 12
MATLAB scripts . . . 13
MATLAB functions . . . 13
Scripts versus functions . . . 14
8.3 Control Structures . . . 14
Loops . . . 14
Conditional statements . . . 15
8.4 Strings and string output . . . 15
9 Debugging your MATLAB code . . . 15

List of Commonly Used MATLAB Commands **19**

10 Useful MATLAB Facts **20**

1 Signals, Signal Statistics, and Signal Detection I **21**
1.1 Introduction . . . 21
1.1.1 "The Questions" . . . 22
1.2 Background . . . 22
1.2.1 Continuous-time and discrete-time signals . . . 22
1.2.2 Describing Signals . . . 23
1.2.3 Signal support and duration . . . 24
1.2.4 Periodicity . . . 24
1.2.5 Signals in MATLAB . . . 25
1.2.6 Signal Statistics . . . 26
1.2.7 Measuring signal distortion and error . . . 28
1.2.8 Signal detection . . . 29
Specifying the detector's operation . . . 29
Detector algorithm . . . 31
1.3 Some MATLAB commands for this lab . . . 32
1.4 Demonstrations in the Lab Session . . . 33
1.5 Laboratory Assignment . . . 34

2 Signal Correlation and Detection II **37**
2.1 Introduction . . . 37
2.1.1 "The Questions" . . . 37
2.2 Background . . . 37
2.2.1 Correlation . . . 37
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 . . . 42
2.2.6 An algorithm for running correlation . . . 44
2.3 Some MATLAB commands for this lab . . . 45

2.4 Demonstrations in the Lab Section . . . 46
2.5 Laboratory Assignment . . . 46

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 . . . 60
3.5 Laboratory Assignment . . . 60

4 Fourier Series and the DFT 65
4.1 Introduction . . . 65
4.1.1 "The Questions" . . . 65
4.2 Background . . . 66
4.2.1 Frequency-domain representations . . . 66
4.2.2 Periodic Continuous-Time Signals — The Fourier Series . . . 66
Partial Series . . . 67
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 . . . 72
Aperiodic Discrete-Time Signals . . . 73
Approximating Fourier series coefficients with the DFT . . . 73
Properties of the DFT coefficients . . . 74
4.2.4 Separating Signals Based on Differing Harmonic Series . . . 76
4.3 Some MATLAB commands for this lab . . . 77
4.4 Demonstrations in the Lab Section . . . 78
4.5 Laboratory Assignment . . . 78

5 Images, Compression, and Coding 83
5.1 Introduction . . . 83
5.1.1 "The Question" . . . 84
5.2 Background . . . 84
5.2.1 Images . . . 84
5.2.2 Signal statistics for images . . . 84
5.2.3 Data compression . . . 86
5.2.4 Transformation . . . 87
5.2.5 Quantization . . . 88
5.2.6 Binary coding . . . 89

5.2.7 Performance . . . 90
5.2.8 Designing a transform coder . . . 92
5.3 Some MATLAB commands for this lab . . . 93
5.4 Demonstrations in the Lab Section . . . 95
5.5 Laboratory assignment . . . 95
Postscript: JPEG Compression . . . 98

6 FIR Filtering and Image Processing 99
6.1 Introduction . . . 99
6.1.1 "The Question" . . . 99
6.2 Background . . . 100
6.2.1 Implementing FIR Filters . . . 100
6.2.2 Edge effects and delay . . . 101
6.2.3 Noise and distortion . . . 102
Nonlinear filtering . . . 103
6.2.4 Filtering two-dimensional signals . . . 104
6.2.5 Image processing with FIR filters . . . 104
6.3 Some MATLAB commands for this lab . . . 106
6.4 Demonstrations in the Lab Section . . . 108
6.5 Laboratory Assignment . . . 109

7 Decoding DTMF: Filters in the Frequency Domain 113
7.1 Introduction . . . 113
7.1.1 "The Question" . . . 113
7.2 Background . . . 114
7.2.1 DTMF signals and Touch Tone™ Dialing . . . 114
7.2.2 Decoding DTMF Signals . . . 114
Step 1: Bandpass Filters . . . 114
Step 2: Determining filter output strengths . . . 116
Step 3: "Detect and Decode" . . . 116
7.2.3 Decoder Robustness . . . 118
7.2.4 Sidenote: Searching Parameter Spaces . . . 118
7.3 Some MATLAB commands for this lab . . . 118
7.4 Demonstrations in the Lab Section . . . 121
7.5 Laboratory Assignment . . . 121

8 Classification and Vowel Recognition 125
8.1 Introduction . . . 125
8.1.1 "The Question" . . . 125
8.2 Background . . . 126
8.2.1 An Introduction to Classification . . . 126
8.2.2 A classification example . . . 127
8.2.3 A few more classification examples . . . 129
8.2.4 Formalizing the feature classifier . . . 130
8.2.5 Measuring the performance of a classifier . . . 132
Data usage when designing classifiers . . . 132
8.2.6 Vowel Classification . . . 133
8.3 Some MATLAB commands for this lab . . . 134

8.4 Demonstrations in the Lab Section . . . 136
8.5 Laboratory Assignment . . . 136

9 Filter Design, Modeling, and the z-Plane 141
9.1 Introduction . . . 141
9.1.1 "The Question" . . . 141
9.2 Background . . . 142
9.2.1 Filters and the z-transform . . . 142
9.2.2 FIR Filters and the z-transform . . . 142
9.2.3 IIR filters and rational system functions . . . 144
Poles and zeros at the origin and at infinity . . . 145
Pole-zero plots . . . 146
9.2.4 Graphical interpretation of the system function . . . 147
9.2.5 Poles and stability . . . 147
9.2.6 Filter design using manual pole-zero placement . . . 149
9.2.7 Design of Standard Filter Types . . . 149
9.2.8 Modeling Vowel Production . . . 150
All-pole analysis and vowel classification . . . 152
9.3 Some MATLAB commands for this lab . . . 153
9.4 Demonstrations in the Lab Section . . . 156
9.5 Laboratory Assignment . . . 156

Contents

List of Figures

1.1 Examples of continuous-time and discrete-time signals . . . 23
1.2 Signal value distribution and a discrete histogram approximation . . . 28
1.3 An "overview" block diagram for a "signal present" detector. . . . 30
1.4 A detailed block diagram for the "signal present" detector. . . . 30

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. . . . 40
2.4 Code signals for a simultaneous communication system . . . 41
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 . . . 75
4.3 The DFT of a harmonic series . . . 76

5.1 A block diagram of a general data compression system. . . . 83
5.2 A block diagram of a general image encoder/compressor. . . . 86
5.3 A block diagram of a general image decoder/decompressor. . . . 87
5.4 Block diagram of a direct quantizer . . . 90
5.5 Plot of the tradeoff between coding rate and distortion . . . 91
5.6 A block diagram of a transform coder . . . 92

6.1 A graphical illustration of filtering . . . 101
6.2 Input and output of a 5-point moving average filter. . . . 102
6.3 A block diagram of a noise-reduction system . . . 103
6.4 The coefficients of a two-dimensional moving average filter . . . 105
6.5 The coefficients for `g_smooth` filters with varying widths. . . . 108

7.1 A spectrogram of a DTMF signal . . . 115
7.2 A block diagram of the DTMF decoder system . . . 116
7.3 A comparison of half-wave and full-wave rectification . . . 117
7.4 An illustration of the DTMF detector subsystem . . . 118

8.1 Block diagram of a general classifier system. . . . 126
8.2 A histogram for one-feature classification . . . 127
8.3 A case where two-feature classification is necessary . . . 128
8.4 An example where two features are not as clearly separated. . . . 129
8.5 Various classification examples . . . 130
8.6 Block diagram of a distance-based feature classifier . . . 131
8.7 The magnitude spectrum (in decibels) of four vowel signals . . . 133

9.1 A pole-zero plot of an IIR filter. . . . 146
9.2 System function, frequency response, and pole-zero plot of an FIR filter . . . 148
9.3 System function, frequency response, and pole-zero plot of an all-pole filter . . . 148
9.4 System function, frequency response, and pole-zero plot of a general IIR filter . . . 148
9.5 An illustration of the various bands of a lowpass filter. . . . 150
9.6 A diagram showing the larynx and vocal tract. . . . 151
9.7 A block diagram of the source-filter model of speech production. . . . 151
9.8 Plot of the spectral evolution of a vowel signal . . . 152
9.9 The GUI window for *Pole-Zero Place 3-D*. . . . 154

Lab 0: 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;
```

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

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
```

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.

[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.


```
>> 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 **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:

```
>> 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:

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


```
>> 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. Also try `zoom xon` for zooming only the horizontal axis.

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.

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

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.

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 19. Included are various helpful pieces of information for working with MATLAB.

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;
    ```

