A typical undergraduate electrical engineering curriculum incorporates a signals and systems course. The widely used approach for the laboratory component of such courses involves the utilization of MATLAB to implement signals and systems concepts. This book presents a newly developed laboratory paradigm where MATLAB codes are made to run on smartphones which are possessed by nearly all students. As a result, this laboratory paradigm provides an anywhere–anytime hardware platform or processing board for students to learn implementation aspects of signals and systems concepts. The book covers the laboratory experiments that are normally covered in signals and systems courses and discusses how to run MATLAB codes for these experiments as apps on both Android and iOS smartphones, thus enabling a truly mobile laboratory paradigm.

A zipped file of the codes discussed in the book can be acquired via the website:
Anywhere-Anytime
Signals and Systems Laboratory
From MATLAB to Smartphones
Third Edition
Synthesis Lectures on Signal Processing

Editor
José Moura, Carnegie Mellon University

Synthesis Lectures in Signal Processing publish 80- to 150-page books on topics of interest to signal processing engineers and researchers. The Lectures exploit in detail a focused topic. They can be at different levels of exposition—from a basic introductory tutorial to an advanced monograph—depending on the subject and the goals of the author. Over time, the Lectures will provide a comprehensive treatment of signal processing. Because of its format, the Lectures will also provide current coverage of signal processing, and existing Lectures will be updated by authors when justified.

Lectures in Signal Processing are open to all relevant areas in signal processing. They will cover theory and theoretical methods, algorithms, performance analysis, and applications. Some Lectures will provide a new look at a well-established area or problem, while others will venture into a brand new topic in signal processing. By careful reviewing the manuscripts we will strive for quality both in the Lectures’ contents and exposition.

Anywhere-Anytime Signals and Systems Laboratory: From MATLAB to Smartphones, Third Edition
Nasser Kehtarnavaz, Fatemeh Saki, Adrian Duran, and Arian Azarang
2020

Reconstructive-Free Compressive Vision for Surveillance Applications
Henry Braun, Pavan Turaga, Andreas Spanias, Sameeksha Katoch, Suren Jayasuriya, and Cihan Tepedelenlioglu
2019

Smartphone-Based Real-Time Digital Signal Processing, Second Edition
Nasser Kehtarnavaz, Abhishek Sehgal, Shane Parris
2018

Anywhere-Anywhere Signals and Systems Laboratory: from MATLAB to Smartphones, Second Edition
Nasser Kehtarnavaz, Fatemeh Saki, and Adrian Duran
2018
Nonlinear Source Separation
Luis B. Almeida
2006

Spectral Analysis of Signals: The Missing Data Case
Yanwei Wang, Jian Li, and Petre Stoica
2006
Anywhere-Anytime
Signals and Systems Laboratory
From MATLAB to Smartphones
Third Edition

Nasser Kehtarnavaz, Fatemeh Saki, Adrian Duran, and Arian Azarang
University of Texas at Dallas

SYNTHESIS LECTURES ON SIGNAL PROCESSING #18
ABSTRACT
A typical undergraduate electrical engineering curriculum incorporates a signals and systems course. The widely used approach for the laboratory component of such courses involves the utilization of MATLAB to implement signals and systems concepts. This book presents a newly developed laboratory paradigm where MATLAB codes are made to run on smartphones which are possessed by nearly all students. As a result, this laboratory paradigm provides an anywhere-anytime hardware platform or processing board for students to learn implementation aspects of signals and systems concepts. The book covers the laboratory experiments that are normally covered in signals and systems courses and discusses how to run MATLAB codes for these experiments as apps on both Android and iOS smartphones, thus enabling a truly mobile laboratory paradigm. A zipped file of the codes discussed in the book can be acquired via the website http://sites.fastspring.com/bookcodes/product/SignalsSystemsBookcodesThirdEdition.

KEYWORDS
smartphone-based signals and systems laboratory, anywhere-anytime platform for signals and systems courses, from MATLAB to smartphones
## Contents

**Preface** ................................................................. xiii

1  **Introduction to MATLAB** ........................................... 1
   1.1 Starting MATLAB .................................................. 1
       1.1.1 Arithmetic Operations .................................... 6
       1.1.2 Vector Operations .......................................... 7
       1.1.3 Complex Numbers ........................................... 8
       1.1.4 Array Indexing .............................................. 8
       1.1.5 Allocating Memory ........................................... 9
       1.1.6 Special Characters and Functions ......................... 10
       1.1.7 Control Flow ................................................ 10
       1.1.8 Programming in MATLAB ................................... 13
       1.1.9 Sound Generation ............................................ 14
       1.1.10 Loading and Saving Data ................................ 15
       1.1.11 Reading Wave and Image Files ............................. 15
       1.1.12 Signal Display ............................................... 16
   1.2 MATLAB Programming Examples .................................. 16
       1.2.1 Signal Generation ........................................... 16
       1.2.2 Generating a Periodic Signal .............................. 19
   1.3 Lab Exercises .................................................... 19

2  **Software Development Tools** ..................................... 23
   2.1 Android Tools Installation Steps ............................... 23
       2.1.1 Java JDK ................................................... 23
       2.1.2 Android Studio Development Environment and Native Development Kit ........................................... 25
       2.1.3 Android Studio Setup Wizard ............................... 25
       2.1.4 Android Emulator Configuration ........................... 27
       2.1.5 Getting Familiar with Android Software Tools .......... 30
   2.2 iOS Tools Installation Steps ..................................... 44
       2.2.1 iPhone App Development with Xcode ...................... 45
       2.2.2 Setting-Up App Environment ............................... 45
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Fourier Series</td>
<td>135</td>
</tr>
<tr>
<td>5.1</td>
<td>Fourier Series Numerical Computation</td>
<td>136</td>
</tr>
<tr>
<td>5.2</td>
<td>Fourier Series and its Applications</td>
<td>137</td>
</tr>
<tr>
<td>5.3</td>
<td>Lab Exercises</td>
<td>144</td>
</tr>
<tr>
<td>5.3.1</td>
<td>RL Circuit Analysis</td>
<td>144</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Doppler Effect</td>
<td>151</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Synthesis of Electronic Music</td>
<td>152</td>
</tr>
<tr>
<td>5.4</td>
<td>Real-Time Labs</td>
<td>154</td>
</tr>
<tr>
<td>5.5</td>
<td>References</td>
<td>175</td>
</tr>
<tr>
<td>6</td>
<td>Continuous-Time Fourier Transform</td>
<td>177</td>
</tr>
<tr>
<td>6.1</td>
<td>CTFT and its Properties</td>
<td>177</td>
</tr>
<tr>
<td>6.2</td>
<td>Numerical Approximations of CTFT</td>
<td>177</td>
</tr>
<tr>
<td>6.3</td>
<td>Evaluating Properties of CTFT</td>
<td>179</td>
</tr>
<tr>
<td>6.4</td>
<td>Lab Exercises</td>
<td>198</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Circuit Analysis</td>
<td>198</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Doppler Effect</td>
<td>204</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Diffraction of Light</td>
<td>204</td>
</tr>
<tr>
<td>6.5</td>
<td>Real-Time Labs</td>
<td>206</td>
</tr>
<tr>
<td>6.6</td>
<td>References</td>
<td>226</td>
</tr>
<tr>
<td>7</td>
<td>Digital Signals and Their Transforms</td>
<td>227</td>
</tr>
<tr>
<td>7.1</td>
<td>Digital Signals</td>
<td>227</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Sampling and Aliasing</td>
<td>227</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Quantization</td>
<td>230</td>
</tr>
<tr>
<td>7.1.3</td>
<td>A/D and D/A Conversions</td>
<td>233</td>
</tr>
<tr>
<td>7.1.4</td>
<td>DTFT and DFT</td>
<td>234</td>
</tr>
<tr>
<td>7.2</td>
<td>Analog-to-Digital Conversion, DTFT and DFT</td>
<td>235</td>
</tr>
<tr>
<td>7.3</td>
<td>Lab Exercises</td>
<td>251</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Dithering</td>
<td>251</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Image Processing</td>
<td>252</td>
</tr>
<tr>
<td>7.3.3</td>
<td>DTMF Decoder</td>
<td>252</td>
</tr>
<tr>
<td>7.4</td>
<td>References</td>
<td>252</td>
</tr>
<tr>
<td>Authors' Biographies</td>
<td>253</td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td>255</td>
<td></td>
</tr>
</tbody>
</table>
A typical undergraduate electrical engineering curriculum incorporates a signals and systems course where students normally first encounter signal processing concepts of convolution, Fourier series, Fourier transform, and discrete Fourier transform. For the laboratory component of such courses, the conventional approach has involved a laboratory environment consisting of computers running MATLAB codes. There exist several lab textbooks or manuals for the laboratory component of signals and systems courses based on MATLAB, e.g., An Interactive Approach to Signals and Systems Laboratory by Kehtarnavaz, Loizou, and Rahman; Signals and Systems Laboratory with MATLAB by Palamides and Veloni; Signals and Systems: A Primer with MATLAB by Sadiku and Ali; and Signals and Systems by Mitra.

The motivation for writing this lab textbook/manual has been to provide an alternative laboratory paradigm to the above conventional laboratory paradigm by using smartphones as a truly mobile anywhere-anytime hardware platform or processing board for students to run signals and systems codes written in MATLAB on them. This approach or laboratory paradigm eases the requirement of using a dedicated laboratory room for signals and systems courses and allows students to use their own computers/laptops and smartphones/tablets as the hardware platform to learn the implementation aspects of signals and systems concepts. It is worth stating that this book is only meant as an accompanying lab book to signals and systems textbooks and is not meant to be used as a substitute for these textbooks.

The challenge in developing this alternative approach has been to limit the programming language required from students to MATLAB and not requiring them to know any other programming language. MATLAB is extensively used in engineering departments and students are often expected to use it for various courses they take during their undergraduate studies.

The above challenge is met here by using the smartphone software tools that are publicly available. The software development environments of smartphones (both Android and iOS) are free of charge and students can download and place them on their own computers/laptops. In this lecture series book, we have developed the software shells that allow students to run MATLAB codes on their own smartphones/tablets as apps. In the first edition of the book, the implementation was done on Android smartphones. In the second edition, in addition to Android smartphones, the implementation was done on iOS smartphones. Due to various updates that have taken place in MATLAB and in smartphone software tools, this third edition is written to address incompatibility errors caused by the older versions of the software tools when running the codes in the previous editions.

The book chapters correspond to the following labs for a semester-long lab course: (1) introduction to MATLAB programming; (2) smartphone development tools (both Andorid
and iOS); (3) use of MATLAB Coder to generate C codes from MATLAB and running C codes on smartphones; (4) linear time-invariant systems and convolution; (5) Fourier series; (6) continuous-time Fourier transform; and (7) digital signals and discrete Fourier transform. A typical signals and systems laboratory course or component covers the labs associated with subjects (4)–(7).

Finally, it is to be noted that the codes discussed in the book can be acquired from this third-party website http://sites.fastspring.com/bookcodes/product/SignalsSystemsBookcodesThirdEdition.

Nasser Kehtarnavaz, Fatemeh Saki, Adrian Duran, and Arian Azarang
Summer 2020
INTRODUCTION TO MATLAB

MATLAB is a programming environment that is widely used to solve engineering problems. There are many online references on MATLAB that one can read to become familiar with this programming environment. This chapter is only meant to provide an overview or a brief introduction to MATLAB. Screenshots are used to show the steps to be taken and configuration options to set when using the Windows operating system.

1.1 STARTING MATLAB

Assuming MATLAB is installed on the laptop or computer used, select MATLAB from the Start bar of Windows, as illustrated in Figure 1.1. After starting MATLAB, a window called MATLAB desktop appears, see Figure 1.2, which contains other sub-windows or panels. The panel Command Window allows interactive computation to be conducted. Suppose it is desired to compute $3 + 4 \times 6$. This is done by typing it at the prompt command denoted by $>>$; see Figure 1.3. Since no output variable is specified for the result of $3 + 4 \times 6$, MATLAB returns the value in the variable ans, which is created by MATLAB. Note that ans is always overwritten by MATLAB, so if the result is used for another operation, it needs to be assigned to a variable, for example $x = 3 + 4 \times 6$.

In practice, a sequence of operations is usually performed to achieve a desired output. Often, a so-called m-file script is used for this purpose. An m-file script is a simple text file where MATLAB commands are listed. Figure 1.4 shows how to start a new script. In the HOME menu, locate the New Script tab under New → Script, or Ctrl+N to create a blank script under the panel EDITOR. When a new script is opened, it looks as shown in Figure 1.5. A script can be saved using a specified name in a desired location. An m-file script is saved with ‘.m’ extension. When such a file is run, MATLAB reads the commands and executes them as though there were the MATLAB commands and operations. The following section provides more details on the MATLAB commands and operations.
Figure 1.1: MATLAB appearance in windows start bar.
Figure 1.2: MATLAB interface window.
4 1. INTRODUCTION TO MATLAB

Figure 1.3: A simple computation in command window.
Figure 1.4: Starting a new m-file script in MATLAB.
1. INTRODUCTION TO MATLAB

1.1.1 ARITHMETIC OPERATIONS

There are four basic arithmetic operators in m-files:

+ addition
- subtraction
* multiplication
/ division (for matrices, it also means inversion)

The following three operators work on an element-by-element basis:

.* multiplication of two vectors, element-wise
./ division of two vectors, element-wise
.^ raising all the elements of a vector to a power

As an example, to evaluate the expression $a^3 + \sqrt{bd} - 4c$, where $a = 1.2$, $b = 2.3$, $c = 4.5$, and $d = 4$, type the following commands in the Command Window to get the answer (ans):

```matlab
a = 1.2;
b = 2.3;
c = 4.5;
d = 4;
an = a^3 + sqrt(b*d) - 4*c;
an
```
>> a=1.2;
>> b=2.3;
>> c=4.5;
>> d=4;
>> a^3+sqrt(b*d)-4*c
ans =
-13.2388

Note the semicolon after each variable assignment. If the semicolon is omitted, the interpreter echoes back the variable value.

### 1.1.2 VECTOR OPERATIONS

Consider the vectors \( \mathbf{x} = [x_1, x_2, \ldots, x_n] \) and \( \mathbf{y} = [y_1, y_2, \ldots, y_n] \). The following operations indicate the resulting vectors:

\[
\begin{align*}
\mathbf{x} \cdot \mathbf{y} & = [x_1 y_1, x_2 y_2, \ldots, x_n y_n] \\
\mathbf{x} / \mathbf{y} & = \left[ \frac{x_1}{y_1}, \frac{x_2}{y_2}, \ldots, \frac{x_n}{y_n} \right] \\
\mathbf{x} \cdot \mathbf{p} & = [x_1^p, x_2^p, \ldots, x_n^p].
\end{align*}
\]

Considering that the boldfacing of vectors/matrices are not used in .m files, in the notation adopted in this book, no boldfacing of vectors/matrices is shown to retain notation consistency with .m files.

The arithmetic operators + and − can be used to add or subtract matrices, vectors, or scalars. Vectors denote 1-dimensional (1-D) arrays and matrices denote multi-dimensional arrays. For example:

\[
\begin{aligned}
\text{>> } & \mathbf{x}=\begin{bmatrix} 1 & 3 & 4 \end{bmatrix} \\
\text{>> } & \mathbf{y}=\begin{bmatrix} 4 & 5 & 6 \end{bmatrix} \\
\text{>> } & \mathbf{x}+\mathbf{y} \\
\text{ans} & =
\begin{bmatrix} 5 & 8 & 10 \end{bmatrix}
\end{aligned}
\]

In this example, the operator + adds the elements of the vectors \( \mathbf{x} \) and \( \mathbf{y} \), element by element, assuming that the two vectors have the same dimension, in this case 1 × 3 or one row with three columns. An error occurs if one attempts to add vectors having different dimensions. The same applies for matrices.

To compute the dot product of two vectors (in other words, \( \sum_i x_i y_i \)), use the multiplication operator \( \cdot \) as follows:
1. INTRODUCTION TO MATLAB

```
>> x*y'
ans =
   43
```

Note the single quote after `y` denotes the transpose of a vector or a matrix.

An element-by-element multiplication of two vectors (or two arrays) is computed by the following operator:

```
>> x .* y
ans =
   4   15   24
```

That is, \( x \cdot y \) means \([1 \times 1, 3 \times 1, 4 \times 1] = [4 \ 15 \ 24] \).

### 1.1.3 COMPLEX NUMBERS

MATLAB supports complex numbers. The imaginary number is denoted with the symbol `i` or `j`, assuming that these symbols have not been used any other place in the program. It is critical to avoid such symbol conflicts for obtaining correct outcomes. Enter the following and observe the outcomes:

```
>> z=3 + 4i % note the multiplication sign `*´ is not needed after 4
>> conj(z) % computes the conjugate of z
>> angle(z) % computes the phase of z
>> real(z) % computes the real part of z
>> imag(z) % computes the imaginary part of z
>> abs(z) % computes the magnitude of z
```

One can also define an imaginary number with any other user-specified variables. For example, in the following manner:

```
>> img=sqrt(-1)
>> z=3+4*img
>> exp(pi*img)
```

### 1.1.4 ARRAY INDEXING

In m-files, all arrays (vectors) are indexed starting from 1; in other words, `x(1)` denotes the first element of the array `x`. Note that arrays are indexed using parentheses `(.)` and not square
brackets \[ . \] , as done in C/C++. To create an array featuring the integers 1–6 as elements, enter:

```matlab
gt x=[1,2,3,4,5,6]
```

Alternatively, the notation `:` can be used as follows:

```matlab
gt x=1:6
```

This notation creates a vector starting from 1–6, in steps of 1. If a vector from 1–6 in steps of 2 is desired, then type:

```matlab
gt x=1:2:6
```

```matlab
ans =
1 3 5
```

Additional examples are listed below:

```matlab
gt ii=2:4:17
gt jj=20:-2:0
gt ii=2:(1/10):4
```

One can easily extract numbers in a vector. To concatenate an array, the example below shows how to use the operator `[ ]`:

```matlab
gt x=[1:3 4 6 100:110]
```

To access a subset of this array, type the following:

```matlab
gt x(3:7)
gt length(x)  \% gives the size of the array or vector
gt x(2:2:length(x))
```

### 1.1.5 ALLOCATING MEMORY

Memory can get allocated for 1-D arrays (vectors) using the command or function `zeros`. The following command allocates memory for a 100-dimensional array:
1. INTRODUCTION TO MATLAB

```matlab
>> y=zeros(100,1);
>> y(30)
ans =
  0
```

The function `zeros(n,m)` creates an $n \times m$ matrix with all 0 elements. One can allocate memory for 2-dimensional (2-D) arrays (matrices) in a similar fashion. The command or function

```matlab
>> y=zeros(4,5)
```

defines a 4 by 5 matrix.

Similar to the command `zeros`, the command `ones` can be used to define a vector containing all ones. For example,

```matlab
>> y=ones(1,5)
ans=
  1 1 1 1 1
```

### 1.1.6 SPECIAL CHARACTERS AND FUNCTIONS

Here is an example of the function `length`:

```matlab
>> x=1:10;
>> length(x)
ans =
  10
```

The function `find` returns the indices of a vector that are non-zero. For example,

```matlab
I = find(x>4)
```

finds all the indices of $x$ greater than 4. Thus, for the above example:

```matlab
>> find(x> 4)
ans =
  5 6 7 8 9 10
```

### 1.1.7 CONTROL FLOW

m-files have the following control flow constructs:
1.1. STARTING MATLAB

Table 1.1: Some widely used special characters used in m-files

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>pi</td>
<td>$\pi$ (3.14.....)</td>
</tr>
<tr>
<td>^</td>
<td>Indicates power (for example, $3^2 = 9$)</td>
</tr>
<tr>
<td>NaN</td>
<td>Not-a-number, obtained when encountering undefined operations, such as $0/0$</td>
</tr>
<tr>
<td>Inf</td>
<td>Represents $+\infty$</td>
</tr>
<tr>
<td>;</td>
<td>Indicates the end of a row in a matrix; also used to suppress printing on the screen (echo off)</td>
</tr>
<tr>
<td>%</td>
<td>Comments—anything to the right of % is ignored by the .m file interpreter and is considered to be comments</td>
</tr>
<tr>
<td>'</td>
<td>Denotes transpose of a vector or a matrix; also used to define string, for example, <code>str1 = 'DSP'</code></td>
</tr>
<tr>
<td>…</td>
<td>Denotes continuation; three or more periods at the end of a line continue current function to next line</td>
</tr>
</tbody>
</table>

Table 1.2: Some widely used functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt</td>
<td>Indicates square root, for example, $\sqrt{4} = 2$</td>
</tr>
<tr>
<td>abs</td>
<td>Absolute value $</td>
</tr>
<tr>
<td>length</td>
<td>$length(x)$ gives the dimension of the array $x$</td>
</tr>
<tr>
<td>sum</td>
<td>Finds sum of the elements of a vector</td>
</tr>
<tr>
<td>find</td>
<td>Finds indices of nonzero</td>
</tr>
</tbody>
</table>

- **if** statements
- **switch** statements
- **for** loops
- **while** loops
- **break** statements

The constructs **if**, **for**, **switch**, and **while** need to terminate with an **end** statement. Examples are provided below:
1. INTRODUCTION TO MATLAB

if

>> x=-3;
if x>0
    str='positive'
elseif x<0
    str='negative'
elseif x== 0
    str='zero'
else
    str='error'
end

See the value of \( str \) after running the above code.

while

>> x=-10;
while x<0
    x=x+1;
end

See the value of \( x \) after running the above code.

for loop

>> x=0;
for j=1:10
    x=x+j;
end

The above code computes the sum of all the numbers from 1–10.

break

With the break statement, one can exit early from a \texttt{for} or a \texttt{while} loop. For example:

>> x=-10;
while x<0
    x=x+2;
    if x = = -2
Table 1.3: Relational operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;=</td>
<td>Less than equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>=</td>
<td>Equal</td>
</tr>
<tr>
<td>~=</td>
<td>Not equal</td>
</tr>
</tbody>
</table>

Table 1.4: Logical operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>NOT</td>
</tr>
</tbody>
</table>

Some of the supported relational and logical operators are listed below.

1.1.8 PROGRAMMING IN MATLAB

Open a new script file as displayed in Figures 1.3 and 1.4. Save it first in a desired directory. Then, write your MATLAB code and press Run button from the EDITOR panel. For instance, to write a program to compute the average (mean) of a vector \( x \), the program should use the vector \( x \) as its input and return the average value. To write this program, follow the steps outlined below.

Type the following in an empty window:

```matlab
x=1:10
L=length(x);
sum=0;
for j=1:L
    sum=sum+x(j);
end
break;
end
end
```

x=1:10
L=length(x);
sum=0;
for j=1:L
    sum=sum+x(j);
14 1. INTRODUCTION TO MATLAB

Figure 1.6: m-file script interactive window after running the program average.

```matlab
end
y=sum/L % y returns the average of x
```

From the EDITOR panel, go to save → Save As and enter average.m for the filename. Then, click on the Run button to run the program. Figure 1.6 shows the MATLAB interactive window after running the program.

1.1.9 SOUND GENERATION

Assuming the computer used has a sound card, one can use the function sound to play back speech or audio files through its speakers. That is, the function `sound(y, FS)` sends the signal in a vector $y$ (with sample frequency $FS$) out to the speaker. Stereo sounds are played on platforms that support them, with $y$ being an N-by-2 matrix.

Type the following code and listen to a 400 Hz tone:
1.1. STARTING MATLAB

```
>> t=0:1/8000:1;
>> x=cos(2*pi*400*t);
>> sound(x,8000);
```

Now generate a noise signal by typing:

```
>> noise=randn(1,8000); % generate 8000 samples of noise
>> sound(noise,8000);
```

The function `randn` generates Gaussian noise with zero mean and unit variance.

1.1.10 LOADING AND SAVING DATA

One can load or store data using the commands `load` and `save`. To save the vector `x` of the above code in the file `data.mat`, type:

```
>> save('data.mat', 'x')
```

To retrieve the data previously saved, type:

```
>> load data
```

The vector `x` gets loaded in memory. To see memory contents, use the command `whos`:

```
>> whos
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Dimension</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1x8000</td>
<td>double array</td>
</tr>
</tbody>
</table>

The command `whos` gives a list of all the variables currently in memory, along with their dimensions and data type. In the above example, `x` contains 8000 samples.

To clear up memory after loading a file, type `clear all` when done. This is important because if one does not clear all the variables, conflicts can occur with other codes using the same variables.

1.1.11 READING WAVE AND IMAGE FILES

In MATLAB, one can read data from different file types (such as .wav, .jpeg, and .bmp) and load them in a vector.

To read an audio data file with .wav extension, use the following command:
1. INTRODUCTION TO MATLAB

```
>> [y,Fs]=audioread('filename')
```

This command reads a .wav file specified by the string `filename` and returns the sampled data in `y` with the sampling rate of `Fs` (in Hz).

To read an image file, use the following command:

```
>> [y]=imread('filename')
```

This command reads a grayscale or color image from the string `filename` and returns the image data in the array `y`.

1.1.12 SIGNAL DISPLAY

Graphical tools are available in MATLAB to display data in a graphical form. Throughout the book, signals in both the time and frequency domains are displayed using the function `plot`.

```
>> plot(x,y)
```

This function creates a 2-D line plot of the data in `y` vs. corresponding `x` values.

1.2 MATLAB PROGRAMMING EXAMPLES

In this section, several simple MATLAB programs are covered.

1.2.1 SIGNAL GENERATION

In this example, let us see how to generate and display continuous-time signals in the time domain. One can represent such signals as a function of time. For simulation purposes, a representation of time `t` is needed. Note that the time scale is continuous while computers handle operations in a discrete manner. Continuous-time simulation is achieved by considering a very small time interval. For example, if a 1-s duration signal in millisecond (ms) increments (time interval of 0.001 s) is considered, then one sample every 1 ms or a total of 1000 samples are generated for the entire signal leading to a continuous signal simulation. This continuous-time signal approximation or simulation is used in later chapters. It is important to note that a finite number of samples is involved in the simulation of a continuous-time signal, and thus to differentiate a continuous-time signal from a discrete-time signal, a much higher number of samples per second for a continuous-time signal needs to be used (very small time interval).
Figure 1.7: Continuous-time signal.

Figure 1.7 shows two continuous-time signals $x_1(t)$ and $x_2(t)$ with a duration of 3 s. By setting the time interval $dt$ to 0.001 s, there is a total of 3000 samples at $t = 0, 0.001, 0.002, 0.003, \ldots, 2.999$ s. Note that throughout the book, the notations $dt$, delta, and $\Delta$ are used interchangeably to denote the time interval between samples.

The signal $x_1(t)$ can be represented mathematically as follows:

\[
x_1(t) = \begin{cases} 
0 & 0 \leq t < 1 \\
1 & 1 \leq t < 2 \\
0 & 2 \leq t < 3.
\end{cases} \quad (1.1)
\]

To simulate this analog or continuous-time signal, use the MATLAB functions `ones` and `zeros`. The signal value is zero during the first second, which means the first 1000 samples are zero. This portion of the signal is simulated with the function `zeros(1,1000)`. In the next second (next 1000 samples), the signal value is 2, and this portion is simulated by the function `2*ones(1,1000)` . Finally, the third portion of the signal is simulated by the function `zeros(1,1000)` . In other words, the entire duration of the signal is simulated by the following .m file function:

\[
x1=[\text{zeros}(1,1/dt) \ 2*\text{ones}(1,1/dt) \ \text{zeros}(1,1/dt)]
\]

The signal $x_2(t)$ can be represented mathematically as follows:

\[
x_2(t) = \begin{cases} 
2t & 0 \leq t < 1 \\
-2t + 4 & 1 \leq t < 2 \\
0 & 2 \leq t < 3.
\end{cases} \quad (1.2)
\]

A linearly increasing or decreasing vector can thus be used to represent the linear portions. The time vectors for the three portions or segments of the signal are $0:dt:1-dt$, $1:dt:2-dt$, and $2:dt:3-dt$. The first segment is a linear function corresponding to a time vector with a slope of 2; the second segment is a linear function corresponding to a time vector with a slope of $-2$.
and an offset of 4; and the third segment is simply a constant vector of zeros. In other words, the entire duration of the signal for any value of $dt$ can be simulated by the following .m file function:

\[
x2=\begin{bmatrix}2*(0:dt:(1-dt)) -2*(1:dt:(2-dt))+4 zeros(1,1/dt)\end{bmatrix}
\]

Figures 1.8 and 1.9 show the MATLAB code and the plot of the above signal generation, respectively. Signals can be displayed using the function `plot(t, data)`. For proper plotting, first the correct `t` vector needs to be generated. Here this is done by using the function `linspace`:

\[
>> t=linspace(0,E,N)
\]
This function generates a vector \( t \) of \( N \) points linearly spaced between and including \( 0 \) and \( E \), where \( N \) is equal to \( E/dt \).

### 1.2.2 GENERATING A PERIODIC SIGNAL

In this example, a simple periodic signal is generated. This example involves generating a periodic signal in textual mode and displaying it graphically. The shape of the signal (\( \text{sin} \), \( \text{square} \), \( \text{triangle} \), or \( \text{sawtooth} \)) can be modified as well as its frequency and amplitude by using appropriate control parameters. The MATLAB code and the plots generated by it are shown in Figures 1.10 and 1.11, respectively.

Now consider an m-file code to generate four types of waveforms using the functions \( \text{sin} \), \( \text{square} \), and \( \text{sawtooth} \). To change the amplitude and frequency of the waveforms, two control parameters named Amplitude (\( A \)) and Frequency (\( f \)) are used. Waveform Type (\( w \)) is another parameter used for controlling the waveform type. With this control parameter, one can select from multiple inputs. Finally, the waveforms are displayed by using the function \( \text{plot} \).

### 1.3 LAB EXERCISES

1. Write an m-file code to add all the numbers corresponding to the even indices of an array. For instance, if the array \( x \) is specified as \( x = [1, 3, 5, 10] \), then 13 (\( = 3 + 10 \)) should be returned. Use the program to find the sum of all even integers from 1–1000. Run your code. Also, rewrite the code where \( x \) is the input vector and \( y \) is the sum of all the numbers corresponding to the even indices of \( x \).
2. Explain the operation performed by the following .m file:

```matlab
L = length(x);
for j = 1:L
    if x(j) < 0
        x(j) = -x(j);
    end
end
```

Rewrite this program without using a `for` loop.

3. Write a .m file code that implements the following hard-limiting function:

\[
x(t) = \begin{cases} 
0.2 & t \geq 0.2 \\
-0.2 & t < 0.2.
\end{cases}
\]  

(1.3)
For $t$, use 1000 random numbers generated by using the function `rand`.

4. Write a MATLAB code to generate two sinusoid signals with the frequencies $f_1$ Hz and $f_2$ Hz and the amplitudes $A_1$ and $A_2$, based on a sampling frequency of 8000 Hz with the number of samples being 256. Set the frequency ranges from 100–400 Hz and set the amplitude ranges from 20–200. Generate a third signal with the frequency $f_3 = \text{mod}(\text{lcm}(f_1, f_2), 400) + 100$ Hz, where `mod` and `lcm` denote the modulus and least common multiple operation, respectively, and the amplitude $A_3$ is the sum of the amplitudes $A_1$ and $A_2$. Use the same sampling frequency and number of samples as specified for the first two signals. Display all the signals using the legend on the same waveform graph and label them accordingly.
CHAPTER 2

Software Development Tools

This chapter covers the steps that need to be taken in order to install the software tools for running C codes on Android and iPhone smartphones. Later in Chapter 3, it will be described how to convert MATLAB codes to C codes.

The Android development environment used here is the IntelliJ IDEA-based Android Studio Bundle (called Android Studio). C codes are made available to the Android Java environment through the use of the Java Native Interface (JNI) wrapper. Thus, it is also necessary to install the Android Native Development Kit (NDK). This development kit allows one to write C codes, compile, and debug them on an emulated Android platform or on an actual Android smartphone/tablet.

Screenshots are used to show the steps and configuration options involved in the installation when using the Windows operating system. The same software tools are also available for other operating systems.

2.1 ANDROID TOOLS INSTALLATION STEPS

This section covers the installation of the necessary software packages for Android app development. Start by creating a directory where the tools are to be installed. A generic directory of \texttt{C:\Android} is used here and the setup is done such that all Android development related files are placed within the directory \texttt{C:\Android}. Before running Android Studio, the latest Java JDK needs to be installed.

2.1.1 JAVA JDK

If the Java Development Kit (JDK) is not already installed on your computer or you do not have the latest version, download it from Oracle's website and follow the installation steps indicated by the installer. The latest JDK package at the time of this writing can be found on the Oracle's website at \texttt{http://www.oracle.com/technetwork/java/javase/downloads/index.html}.

Click on the JDK \texttt{Download} button in the section \texttt{Oracle JDK} as shown in Figure 2.1 and you will be directed to the page shown in Figure 2.2. From the list of supported platforms, select the correct version for your operating system. For example, if you are running a 64-bit operating system, select the appropriate package.
24 2. SOFTWARE DEVELOPMENT TOOLS

Figure 2.1: Java installation.

Figure 2.2: Java SE development kit 14.
2.1. ANDROID TOOLS INSTALLATION STEPS

2.1.2 ANDROID STUDIO DEVELOPMENT ENVIRONMENT AND NATIVE DEVELOPMENT KIT

The most recent versions of Android Studio and the NDK at the time of this writing are used to run the lab experiments in the book. For the Windows, Mac, and Linux installation, the Android Studio is available as an executable installer which includes the development environment. For Android SDK tools, you need to download it separately. First, go the link https://developer.android.com/studio#downloads to see Figure 2.3. Then, click on the appropriate download option shown in this figure. A separate section of the same page gets opened. Find the section Common Line Tools Only and download the SDK for Windows platform. The NDK can be downloaded and installed from within Android Studio as described at http://developer.android.com/studio/index.html.

Download the Android Studio installation executable and run the Android Studio installer. For platform specific instructions, the installation instructions appear at https://developer.android.com/studio/install.html.

During the installation of Android Studio, there are two important settings that are critical to be performed correctly; see Figures 2.4 and 2.5. For the setting shown in Figure 2.4, make sure that all the components are selected for installation, and for the setting shown in Figure 2.5, make sure that Android Studio and Android SDK are installed in the directory C:\Android. To do so, manually create the directories by using the Browse option and create a Studio folder and an sdk folder. When the installer is finished, Android Studio can get started. Then, the Android Studio Setup Wizard is to be activated which is covered next. The Android NDK can be downloaded and installed from within Android Studio.

2.1.3 ANDROID STUDIO SETUP WIZARD

When Android Studio completes its installation, make sure the checkbox to run Android Studio is checked. The Android Studio Setup Wizard begins. Follow the steps noted as follows.
26  2. SOFTWARE DEVELOPMENT TOOLS

Figure 2.4: Android Studio setup.

Figure 2.5: Configuration settings.
• Hit next on the Welcome Screen.

• Select Custom to finish the configuration, as displayed in Figure 2.6.

• Select your preferred UI Theme IntelliJ or Darcula.

• Make sure all boxes are checked under Android SDK, Performance (HAXM), and Android Virtual Device (AVD).
  – Make sure to select the SDK Location as noted above, that is C:\Android\sdk.

• Use the recommended Emulator Settings.

• Review your settings and hit Finish.

When the above is done, the Android Studio home screen should appear, as shown in Figure 2.8.

Now run the SDK Manager, whose entry can be found by clicking on the Configure option. From this menu, additional system images for emulation and API packages for future Android versions can get added. Select any Android API level you may require listed on the SDK Platform tab; see Figure 2.9. On the SDK Tools tab, check the CMake, LLDB, NDK (Side by side) tools; see Figure 2.10. Click on Apply and follow the steps.

Allow the update process to complete.

2.1.4 ANDROID EMULATOR CONFIGURATION

The last item to take care of is configuring an AVD by clicking again on the Configure tab shown in Figure 2.8 and select the AVD Manager to open the AVD Manager shown in Figure 2.11. By default, Android Studio creates an x86 AVD. Since our development focus is ARM-based
2. SOFTWARE DEVELOPMENT TOOLS

Figure 2.8: Android Studio home screen.

Figure 2.9: SDK manager.
2.1. ANDROID TOOLS INSTALLATION STEPS

Figure 2.10: SDK tools.

Figure 2.11: Android virtual device.
2. SOFTWARE DEVELOPMENT TOOLS

Figure 2.12: Device selection.

implementation, an ARM-based emulator instance is needed. Begin by deleting any existing AVD instances.

Click the button Create Virtual Device... to start configuring the AVD (see Figure 2.11). For the Device option shown in Figure 2.12, select a device with a good screen resolution for your computer—Nexus S is usually fine. For compatibility with newer smartphones, it is suggested to select the Target as the latest available version with the CPU/ABI as ARM (armeabi-v7a). After selecting a proper device, you will be directed to Verify Configuration shown in Figure 2.13. Using the settings Show Advanced Settings, some changes can be made in the virtual device as illustrated in Figure 2.14. RAM allocation does not need to be large so choose 512 MB. Lastly, set the SD Card and Internal Storage size to 256 MB. The Snapshot option is useful to select as it normally takes a long time for the AVD to boot. The snapshot will save the memory state of the emulator to your hard drive so that starting the emulator occurs much faster. For the first time, it is recommended to select Boot option as Cold boot to configure the virtual device faster.

You should now be able to create the AVD by clicking OK. Select the AVD you just created in the list of devices and click the Start option (see Figure 2.15). The AVD for the created device appears as shown in Figure 2.16.

2.1.5 GETTING FAMILIAR WITH ANDROID SOFTWARE TOOLS

This lab covers the creation of a simple app on Android smartphones by constructing a “Hello World” program. Android Studio and NDK tools are used for code development, emulation, and code debugging. All the codes needed for this and other labs can be extracted from the book
Figure 2.13: Virtual device settings.

Figure 2.14: Advanced settings for virtual device configuration.
codes package mentioned in the preface. Start by launching Android Studio, and if not already done, set up an AVD for use with the Android emulator.

- Begin by creating a new Android project using the Quick Start menu found on the Android Studio home screen.
- Set the Application Name to HelloWorld and the project location to a folder within the directory C:\Android.
- Change the Package name to utdallas.edu.helloworld. This is of importance later as it will affect the naming of your native code methods. Refer to Figure 2.17 for the previous three steps.
- Click Next and on the following screen, choose to create an Empty Activity. The Target Android device should be set to Phone and Tablet using a Minimum SDK setting of API 15.
- Select Finish. The new app project is now created and the main app editor will open to show the GUI layout of the app.
Navigate to the `java` directory of the app in the `Project` window and open the `MainActivity.java` file under `utdallas.edu.helloworld`.

The entity that typically defines an Android app is called an Activity. Activities are generally used to define user interface elements. An Android app has activities containing various sections that users might interact with such as the main app window. Activities can also be used to construct and display other activities—such as if a settings window is needed. Whenever an Android app is opened, the `onCreate` function or method is called. This method can be regarded as the “main” (*C* terminology) of an activity. Other methods may also be called during various portions of the app lifecycle as detailed at the following website:

http://developer.android.com/training/basics/activity-lifecycle/starting.html

In the default code created by the SDK, `setContentView(R.layout.activity_main)` exhibits the GUI. The layout is described in the file `res/layout/activity_main.xml` in the `Package Explorer` window. Open this file to preview the user interface. Layouts can be modified using
the WYSIWYG editor which is built into Android Studio. For now the basic GUI suits our purposes with one minor modification noted below:

- Open the XML text of the layout (see Figure 2.18) by double clicking on the Hello world! text or by clicking on the activity_main.xml tab next to the Graphical Layout tab.

- Add the line android:id="@+id/Log" within the <TextView/> section on a new line and save the changes. This gives a name to the TextView UI element.

TextView in the GUI acts similar to a console window. It displays text. Additional text can be appended by adding the android:id directive to the TextView code.

After setting up the emulator and the app GUI, let us cover interfacing with C codes. Note that it is not required to know the Java code syntax. The purpose is to show that the Java Native Interface (JNI) is a bridge between Java and C codes. Java is useful for handling Android APIs for sound and video i/o, whereas the processing codes are done in C. Note that to conduct the labs in this book, the programming is not done in C and C codes are generated by the MATLAB Coder discussed in the next chapter.

Figure 2.17: Initial screen of Android Studio.
A string returned from a C code is examined next. The procedure to integrate native code consists of creating a C code segment and performing alterations to the project. First, it is required to add support for the native C code to the project. The first step is to create a folder in which the C code will be stored. In the Project listing, navigate to New → Folder → JNI to create a folder in the listing called jni. Refer to Figures 2.19–2.22. Figure 2.19 shows how the Project listing view may be changed in order to show the jni folder in the main source listing.

Android Studio now needs to be configured to build a C code using the Gradle build utility. Begin by specifying the NDK location in the project local.properties file according to Figure 2.23. Assuming the directory C:/Android is used for setting up the development tools, the location specification would be as follows:

```
ndk.dir=C:\\Android\\sdk\\ndk\\21.0.6113669
```

Next, the native library specification needs to get added to the build.gradle file within the project listing under the “app” folder. This specification declares the type of external build and its configuration file which defines the name of the external library that Java will load. This is done by adding the following code within the android section:

```
externalNativeBuild {
    cmake {
        path "CMakeLists.txt"
    }
}
```
Figure 2.19: JNI folder.
Figure 2.20: New Android activity.

Figure 2.21: Project listing.
Figure 2.22: MainActivity.java.

Figure 2.23: local.properties.
The correct placement of the code is highlighted in Figure 2.24. Another part that needs to get added is a `CMakeLists.txt` file in the project section and app folder as shown in Figure 2.25 and by having the following code inside the .txt file:

```cmake
cmake_minimum_required(Versions 3.4.1)
# Creates and names a library, sets it as either STATIC
# or SHARED, and provides the relative paths to its source code.
# You can define multiple libraries, and CMake builds them for you.
# Gradle automatically packages shared libraries with your APK.
add_library(# Sets the name of the library.
    HelloWorld
    # Sets the library as a shared library.
    SHARED
    # Provides a relative path to your source file(s).
    src/main/jni/HelloWorld.c
)
```

The C code considered here consists of a simple method to return a string when it is called from the `onCreate` method. First, the code that defines the native method needs to be included. Create a new `HelloWorld.c` file. Add the following code and save the changes:

```c
#include <jni.h>
jstring Java_utdallas_edu_helloworld_MainActivity_getString
(JNIEnv* env, jobject thiz)
{
    return (*env)->NewStringUTF(env, "Hello UTD!");
}
```

This code defines a method that returns a Java string object according to the JNI specifications with the text Hello UTD! The naming for this method is dependent on what is called fully qualified name of the native method which is defined in MainActivity. There are alternate methods of defining native methods that will be discussed in later labs.

Next, the native method needs to be declared within `MainActivity.java` (see Figure 2.26) according to the naming used in the C code. To do so, add this declaration below the `onCreate` method already defined.

```java
public native String getString();
```

Then, add the following code within `public class` to load the native library:
2. SOFTWARE DEVELOPMENT TOOLS

Figure 2.24: Code placement.

Figure 2.25: CMakeLists placement.