Appendix - A

Introduction to MATLAB®

(The Language of Technical Computing)
A.1 Introduction to MATLAB:

MATLAB® (Matrix Laboratory) is a fourth-generation high performance, high-level programming language for technical computing. It has an interactive environment for numerical computation, and visualization which is an easy-to-use environment where problems and solutions are expressed in familiar with mathematical notation. MATLAB is a technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation. Using the MATLAB product, we can solve technical computing problems faster than the other traditional programming languages, like C, C++, and FORTRAN.

We can use MATLAB in lot of applications, including signal and image processing, communications, control design, test and measurement, financial modeling and analysis, and computational biology. Add-on toolboxes (collections of special-purpose MATLAB functions, available separately) extend the MATLAB environment to solve particular classes of problems in these application areas.

A.2 MATLAB Features:

MATLAB provides a number of features for documenting and sharing our work. We can integrate our MATLAB code with other languages or applications, and distribute our MATLAB algorithms and applications. MATLAB features include:

- High-level language for technical computing
- Development environment for managing code, files, and data
- Interactive tools for iterative exploration, design, and problem solving
• Mathematical functions for linear algebra, statistics, Fourier analysis, filtering, optimization, and numerical integration
• 2-D and 3-D graphics functions for visualizing data
• Tools for building custom graphical user interfaces
• Functions for integrating MATLAB based algorithms with external applications and languages, such as C, C++, Fortran, Java™, COM, and Microsoft® Excel

A.3 The MATLAB System:

The MATLAB system consists of these main parts:

A.3.1 Desktop Tools and Development Environment

This part of MATLAB is the set of tools and facilities that help you use and become more productive with MATLAB functions and files. Many of these tools are graphical user interfaces. It includes: the MATLAB desktop and Command Window, an editor and debugger, a code analyzer, and browsers for viewing help, the workspace, and folders.

A.3.2 Mathematical Function Library

This library is a vast collection of computational algorithms ranging from elementary functions, like sum, sine, cosine, and complex arithmetic, to more sophisticated functions like matrix inverse, matrix eigenvalues, Bessel functions, and fast Fourier transforms.

A.3.3 The Language

The MATLAB language is a high-level matrix/array language with control flow statements, functions, data structures, input/output, and object-oriented
programming features. It allows both "programming in the small" to rapidly create quick programs you do not intend to reuse. You can also do "programming in the large" to create complex application programs intended for reuse.

A.3.4 Graphics

MATLAB has extensive facilities for displaying vectors and matrices as graphs, as well as annotating and printing these graphs. It includes high-level functions for two-dimensional and three-dimensional data visualization, image processing, animation, and presentation graphics. It also includes low-level functions that allow you to fully customize the appearance of graphics as well as to build complete graphical user interfaces on your MATLAB applications.

A.3.5 External Interfaces

The external interfaces library allows you to write C/C++ and Fortran programs that interact with MATLAB. It includes facilities for calling routines from MATLAB (dynamic linking), for calling MATLAB as a computational engine, and for reading and writing MAT-files.

A.4 Uses of MATLAB:

Now a days MATLAB is widely used as a technical computational tool in almost all industries, research and development areas of science and engineering fields. There are approximately 60 toolboxes in MATLAB R2010a version out of which few most important which were used in my research work are as under.
A.4.1 Image Acquisition Tool Box:

The Image Acquisition Toolbox software is a collection of functions that extend the capability of the MATLAB numeric computing environment. The toolbox supports a wide range of image acquisition operations, including:

- Acquiring images through many types of image acquisition devices, from professional grade frame grabbers to USB-based webcams
- Viewing a preview of the live video stream
- Triggering acquisitions (includes external hardware triggers)
- Configuring callback functions that execute when certain events occur
- Bringing the image data into the MATLAB workspace

A.4.2 Image Processing Tool Box:

The Image Processing Toolbox software is a collection of functions that extend the capability of the MATLAB numeric computing environment. The toolbox supports a wide range of image processing operations, including

- Spatial image transformations
- Morphological operations
- Neighborhood and block operations
- Linear filtering and filter design
- Transforms
- Image analysis and enhancement
- Image registration
- Deblurring
- Region of interest operations
A.4.3 Neural Network Tool Box:

Neural networks are composed of simple elements operating in parallel. These elements are inspired by biological nervous systems. As in nature, the connections between elements largely determine the network function. We can train a neural network to perform a particular function by adjusting the values of the connections (weights) between elements.

Typically, neural networks are adjusted, or trained, so that a particular input leads to a specific target output. The figure illustrates such a situation. There, the network is adjusted, based on a comparison of the output and the target, until the network output matches the target. Typically, many such input/target pairs are needed to train a network.

![Artificial neural network working model](image)

Figure A.1: Artificial neural network working model

Neural networks have been trained to perform complex functions in various fields, including pattern recognition, identification, classification, and speech, vision, and control systems.
Neural networks can also be trained to solve problems that are difficult for conventional computers or human beings. The toolbox emphasizes the use of neural network paradigms that build up to--or are themselves used in-- engineering, financial, and other practical applications.

**A.4.4 Optimization Tool Box:**

Optimization Toolbox provides widely used algorithms for standard and large-scale optimization. These algorithms solve constrained and unconstrained continuous and discrete problems. The toolbox includes functions for linear programming, quadratic programming, binary integer programming, nonlinear optimization, nonlinear least squares, systems of nonlinear equations, and multi-objective optimization. We can use them to find optimal solutions, perform tradeoff analyses, balance multiple design alternatives, and incorporate optimization methods into algorithms and models.

Code for the building-block functions is open and extensible. Use the MATLAB Editor to review, copy, and edit code for any function. Extend the toolbox by copying code to new files or by writing files that call toolbox functions.

**A.5 MATLAB Desktop:**

MATLAB is a programming language and a development environment for matrix-based computation. It integrates an editor, an interpreter, and numerous visualization tools in a single, powerful application. MATLAB can be extended with toolboxes, which implement various algorithms commonly used in science and engineering. Of particular interest to us here is the Neural Network Toolbox, which constitutes one of the most comprehensive neural network packages currently
available. While it is quite easy to create and run neural networks within this toolbox, some elementary knowledge of MATLAB is necessary. We have used MATLAB version R2010a in our research work. The IDE of this MATLAB version is shown in figure A.2.

Figure A.2: User Interface of MATLAB

This desktop has four parts.

1. Current Folder
2. Command Window
3. Workspace

- **Current Folder**: It shows the current directory where our files, variables etc. are available, so that we can open any file or load any variable in workspace.
- Command Window: It is used to give any command as per need which is always indicated by prompt (>>). Here we can use any variable or functions.

- Workspace: It is the area where our variables and data files can be imported or export for use.
Command History: It stores the recent executed commands, statements or functions.
A.6 Images in MATLAB:

MATLAB stores most images as two-dimensional arrays, in which each element of the matrix corresponds to a single pixel in the displayed image. For example, an image composed of 200 rows and 300 columns of different colored dots would be stored in MATLAB as a 200-by-300 matrix. Some images, such as RGB, require a three-dimensional array, where the first plane in the third dimension represents the red pixel intensities, the second plane represents the green pixel intensities, and the third plane represents the blue pixel intensities.

This convention makes working with images in MATLAB similar to working with any other type of matrix data, and renders the full power of MATLAB available for image processing applications. For example, a single pixel can be selected from an image matrix using normal matrix subscripting.

```matlab
>> I(25,145)
```

This command will return the value of the pixel at row 25, column 145 of the image which is stored in variable `I`.

MATLAB supports the following graphics file formats:

- BMP  (Microsoft Windows Bitmap)
- HDF   (Hierarchical Data Format)
- JPEG  (Joint Photographic Experts Group)
- PCX   (Paintbrush)
- PNG   (Portable Network Graphics)
- TIFF  (Tagged Image File Format)
- XWD   (X Window Dump)
A.7 Images Processing Function:

As my research work is related to static image processing, here in this section I am describing almost all those image processing functions which are used in my research work. The images which I am using are scanned RGB 26 English alphabets. I have to read all these images in MATLAB and a lot of preprocessing task like conversion in grey image, binary image, edging, dilation etc. have to done. For all these tasks I am using image processing functions which are as below:

A.7.1 imread( ):

The imread(filename) function is used to read a grayscale or color image from the file specified by the filename. This image file should be in current working directory. If the file is not in the current working folder, then we have specify the full path of image with its name and extension. The syntax of this function is as under:

\[
\text{var} = \text{imread}(\text{filename});
\]

For example, I am reading a jpg file “rr.jpg” which is in my current working directory as below

\[
\text{>> w=imread ('rr.jpg');}
\]

There are two things to note about this command:

1. It ends with a semicolon; this has the effect of not displaying the results of the command to the screen. As the result of this particular command is a matrix of size or with elements, we don't really want all its values displayed.
2. The name \texttt{rr.jpg} is given in single quote marks. Without them, MATLAB would assume that \texttt{rr.jpg} was the name of a \texttt{variable}, rather than the name of a file.
A.7.2 `imshow( )`: 

This function is used to display the image stored in variable which could be either RGB or greyscale or binary or any indexed image. But before display any of these image, we should read the image first in any variable. The syntax of this function is:

```
imshow(variable); 
```

For example, I am displaying a jpg image file “rr.jpg” which is stored in w variable using `imread()` function.

```
>> imshow(w) 
```

![Figure A.7: Displaying image using imshow() function](image)

A.7.3 `rgb2gray( )`: 

This function is used to convert a RGB image or a colormap to grayscale intensity image. It converts RGB images to grayscale by eliminating the hue and saturation information while retaining the luminance and store it in another variable. The general syntax of this function is as under:

```
var = rgb2gray(RGB_image)
```
For example, I am converting the RGB image which is stored in variable w to grayscale image using this function as below

```matlab
>> w1 = rgb2gray(w);
```

Now `w1` has a grayscale image to which we can show using `imshow()` function as under.

```matlab
>> imshow(w1)
```

![Figure A.8: Display image in grayscale using rgb2gray() function.](image)

### A.7.4 `imcrop()`:

Using this tool we can crop our image as per our needed for further processing. It creates an interactive Crop Image tool associated with the image displayed in the current figure. This tool is a moveable; resizable rectangle that we can position interactively using the mouse. When the Crop Image tool is active, the pointer changes to cross hairs + when you move it over the target image. Using the mouse, we specify the crop rectangle by clicking and dragging the mouse.

When we finished sizing and positioning; the crop rectangle creates the cropped image by double-clicking the left mouse button or by choosing **Crop Image** from the context menu. The cropped images will returns to the variable for which I am using it.
The syntax is: \[
\text{var=imcrop(image\_file)};
\]

The following example and figure A.9 illustrates the Crop Image tool with the context menu displayed.

\[>>\text{imcrop(w)}\]

Figure A.9: Image Cropping using imcrop( ) function

**A.7.5 im2bw( ):**

This function is used to covert an image form grayscale to binary format. If the input image is not a grayscale image, this function converts the input image to grayscale, and then converts this grayscale image to binary by thresholding.

The syntax of this function is:

\[
\text{BW = im2bw(I, level)};
\]

It will convert the grayscale image \(I\) to a binary image. The output image \(BW\) replaces all pixels in the input image with luminance greater than level with the value 1 (white) and replaces all other pixels with the value 0 (black). Specify level in the range \([0, 1]\). This range is relative to the signal levels possible for the image's class.
Therefore, a level value of 0.5 is midway between black and white. To compute the
level argument, we can use the function `graythresh`. If we do not specify level,
`im2bw()` uses the value 0.5.

Example: Now our grayscale image w1 will be stored as binary image in variable
wb, and we can show it using `imshow()` as shown in figure A.10 below:

```matlab
>>wb=im2bw(w1);
>>imshow(wb);
```

![Figure A.10: Display image in black & white using im2bw() function](image)

**A.7.6 edge ():**

This function is used to detect the edge of an image. The image should be in
greyscale or binary form. As to calculate features of the image, we have to covert the
image form RGB to grayscale, then to binary format and finally find the edges using
`edge()` function. The general syntax is as under:

```
BW = edge(I);
```

Here `I` is a grayscale or a binary image as its input, and it returns a binary
image BW of the same size as of `I`, with 1's where the function finds edges in I and
0's elsewhere. By default, edge uses the *Sobel* method to detect edges. There are
other methods like Prewitt, Roberts, Laplacian of Gaussian, Zero-Cross and Canny which are supported by this function to detect edge. These are as under

\[
A = \text{imread}('1.JPG'); \quad \% \text{Reading rgb image}
\]

\[
A = \text{rgb2gray}(A); \quad \% \text{Converting to grayscale image}
\]

\[
A = \text{im2bw}(A, \text{graythresh}(A)); \quad \% \text{Converting in binary image}
\]

\[
AX = \text{edge}(\text{uint8}(A)); \quad \% \text{Finding Edge of image}
\]

\[
\text{imshow}(AX);
\]

Now the result of imshow(AX) function after edge ( ) function will be like as shown in figure below.

![Figure A.91: Display image in after edging.](image)

**A.7.7 imdilate( ):**

It is basically a mathematical morphological operation to which we can apply on binary and grayscale images. The basic effect of the operator is done on a binary image to gradually enlarge the boundaries of regions of foreground pixels (i.e. white pixels, typically). Thus areas of foreground pixels grow in size while holes within those regions become smaller.
So the imdilate(IM,SE) will dilates the grayscale or binary or packed binary image IM, and will return the dilated image. The argument SE is a structuring element object, or array of structuring element objects, returned by the \texttt{strel} function.

\begin{verbatim}
se = strel('square', 2);
AX = imdilate(AX, se);
imshow(AX)
\end{verbatim}

Now the result of imshow (AX) function after imdilate ( ) function will be like as shown in figure A.12 below.

![Figure A.102: Display image after dilation.](image)

A.7.8 \texttt{imresize()} :

This function is used to resize the image. As I am using 26 images of English alphabets, so I have to make same size of all 26 images so that I could make a pattern of all same size images for further processing.
imresize (A, [mrows ncols]) returns an image that has the number of rows and columns specified by [mrows ncols]. Either NUMROWS or NUMCOLS may be NaN, in this case imresize computes the number of rows or columns automatically to preserve the image aspect ratio.

The syntax is:

\[ \text{Var} = \text{imresize(imagename,[new size])}; \]

For example:

\[
\text{AXR} = \text{imresize(AX,[30 30])};
\]

\[
\text{imshow(AXR)}
\]

Now the result of imshow(AXR) function after imresize ( ) function will be like as shown in figure below. Now the image AXR size converted into 30X30.

![Figure A.13: Display image after resizing to 30X30.](image)

**A.7.9 reshape():**

The reshape function changes the order of matrix with same elements. It produces a matrix with elements taken column by column from the given matrix. For Example: C is a matrix of size 4X5.

\[
\text{AXR} = \text{imresize(AX,[30 30])};
\]

\[
\text{imshow(AXR)}
\]

Now the result of imshow(AXR) function after imresize ( ) function will be like as shown in figure below. Now the image AXR size converted into 30X30.

![Figure A.13: Display image after resizing to 30X30.](image)

**A.7.9 reshape():**

The reshape function changes the order of matrix with same elements. It produces a matrix with elements taken column by column from the given matrix. For Example: C is a matrix of size 4X5.

\[
\text{C} = [1 \ 2 \ 3 \ 4 \ 5 ; \ 6 \ 7 \ 8 \ 9 \ 10 ; \ 11 \ 12 \ 13 \ 14 \ 15 ; \ 16 \ 17 \ 18 \ 19 \ 20 ];
\]

- (182)
C =

1  2  3  4  5  
6  7  8  9 10
11 12 13 14 15
16 17 18 19 20

>> reshape(C, 5, 4)

ans=

1   7  13  19
6  12  18   5
11 17  4  10
16  3   9  15
 2   8  14  20

I am using this function to change the shape of my all 26 images which are in size 30 × 30 to 900 × 1. So that using all 26 images I will make a pattern of 900 × 26 to store in Hopfield Neural Network for further processing.

A.7.10 cell2mat ( ) :

There may be a situation when we are making a array in which each cell itself contained an array. Such arrays are also known as cell type array. To convert this cell array of matrices to single matrix, we can use function \texttt{cell2mat ( )}.

Syntax; \hspace{1cm} m = \texttt{cell2mat(c)} .

It will convert a multidimensional cell array c with contents of the same data type into a single matrix m. The contents of c must be able to concatenate into a hyperrectangle. Moreover, for each pair of neighboring cells, the dimensions of the cells' contents must match, excluding the dimension in which the cells are neighbors.
The results of cell2mat are undefined if your cell array does not adhere to these guidelines.

The example which is shown below combines matrices in a 3-by-2 cell array into a single 60-by-50 matrix:

![Figure A.114: Cell2mat function to make single matrix](image)

**A.8 Neural Network Function:**

Neural networks are composed of simple elements called neurons which operate in parallel. These elements are inspired by biological nervous systems. As in nature, the network function is determined largely by the connections between elements. We can train a neural network to perform a particular function by adjusting the values of the connections (weights) between elements.

Commonly neural networks are adjusted, or trained, so that a particular input leads to a specific target output. So the network is adjusted, based on a comparison of the output and the target, until the network output matches the target.

Neural Network Toolbox is used for designing, implementing, visualizing, and simulating neural networks. Neural networks are invaluable for applications
where formal analysis would be difficult or impossible, such as pattern recognition and nonlinear system identification and control. Neural Network Toolbox software provides comprehensive support for many proven network paradigms, as well as graphical user interfaces (GUIs) that enable us to design and manage your networks.

As in my research work, I am using Hopfield neural network, self-organizing map (SOM), Fast Fourier Transformation (FFT), Inverse of FFT for image pattern storage, various learning methods etc. For all these tasks I am using below neural network functions:

**A.8.1 newhop( ):**

A Hopfield network consists of N fully connected neurons which are both input, functional and output units. It usually process encoded data, i.e. vectors with components in \{-1,1\}. Therefore the typical activation function is signum (satlins in Matlab). A Hopfield network can be created in Matlab by using the function newhop(data) where data is a matrix containing on its rows the data to be stored. This Hopfield network is used for pattern recall.

The general syntax is as below:

\[
\text{net} = \text{newhop}(\text{T})
\]

The newhop (T) function takes one input argument T, which is a R x Q matrix of Q target vectors (values must be +1 or -1) and returns a new Hopfield recurrent neural network with stable points at the vectors in T.

The network functioning is simulated using the function sim. There are two variants of calling the function sim:

\[
\text{result} = \text{sim}(\text{net}, \text{M}, [], \text{test})
\]
or

\[ \text{result} = \text{sim(net, } \{M, \text{ iterations}\}, \{\}, \text{ test}) \]

where M is the number of test data to be taken from the test matrix (specified as the last parameter). In the first variant the user does not control the number of iterations while in the second case user can specify this.

### A.8.2 fft() & ifft():

**Fast Fourier transform & inverse Fast Fourier transform:**

The terms Fast Fourier Transform (FFT) and Inverse Fast Fourier Transform (IFFT) are used to denote efficient and fast algorithms to compute the Discrete Fourier Transform (DFT) and the Inverse Discrete Fourier Transform (IDFT) respectively. The FFT/IFFT is widely used in many digital image / signal processing applications and the efficient implementation of the FFT/IFFT is in continuous research.

A **Fast Fourier transform (FFT)** is a fast computational algorithm to compute the discrete Fourier transform (DFT). The Fast Fourier Transform does not refer to a new or different type of Fourier transform. It refers to a very efficient algorithm for computing the DFT. The FFT utilizes some clever algorithms to do the same thing as the DTF, but in much less time. The general syntax of fft() function is as below:

\[ Y = \text{fft}(X); \]

It will return the discrete Fourier transform (DFT) of vector X, computed with a fast Fourier transform (FFT) algorithm.
• If X is a matrix, fft returns the Fourier transform of each column of the matrix.

• If X is a multidimensional array, fft operates on the first nonsingleton dimension.

The general syntax of ifft() function is as below:

\[ y = \text{ifft}(X) \]

It returns the inverse discrete Fourier transform (DFT) of vector X, computed with a fast Fourier transform (FFT) algorithm. If X is a matrix, ifft returns the inverse DFT of each column of the matrix.

• ifft tests X to see whether vectors in X along the active dimension are conjugate symmetric.

• If so, the computation is faster and the output is real. An N-element vector x is conjugate symmetric if \( x(i) = \text{conj}(x(\text{mod}(N-i+1,N)+1)) \) for each element of x.

• If X is a multidimensional array, ifft operates on the first non-singleton dimension.

A.8.3 newsom( ):

The function newsom() is used to create a Kohonen network (self-organizing map). The general syntax of newsom() function is as below:

\[ \text{mysom} = \text{newsom}(\text{inputData} , [d1,d2,…], \text{tpl, dist, LR1, epochs1, LR2, ns}) \]

Only the first parameter is mandatory. It denotes the set of input data (each data in the set is a column in the matrix inputData).
A.8.4 `som_make()`:

This function is used to creates, initializes and trains a self-organizing map (SOM) using default parameters. The general syntax is as under.

\[
sMap = \text{som\_make}(D);
\]

The `som_make()` uses functions `SOM\_TOPOL\_STRUCT`, `SOM\_TRAIN\_STRUCT`, `SOM\_DATA\_STRUCT` and `SOM\_MAP\_STRUCT` to come up with the default values. First, the number of map units is determined. Unless they are explicitly defined, function `SOM\_TOPOL\_STRUCT` is used to determine this.

After the number of map units has been determined, the map size is determined. Basically, the two biggest eigenvalues of the training data are calculated and the ratio between sidelengths of the map grid is set to this ratio. The actual sidelengths are then set so that their product is as close to the desired number of map units as possible.

Then the SOM is initialized. First, linear initialization along two greatest eigenvectors is tried, but if this can't be done (the eigenvectors cannot be calculated), random initialization is used instead. After initialization, the SOM is trained in two phases: First rough training and second fine-tuning. If the 'tracking' argument is greater than zero, the average quantization error and topographic error of the final map are calculated.

A.8.5 `plotregression()`:

It is used to plot the linear regression of targets relative to actual outputs. For Example: See the below code:
load simplefit_dataset

net = newff(simplefitInputs, simplefitTargets, 20);

[net, tr] = train(net, simplefitInputs, simplefitTargets);

simplefitOutputs = sim(net, simplefitInputs);

plotregression(simplefitTargets, simplefitOutputs);

Figure A.125: plotregression for expected output and actual output.

A.8.6 plotsompos( ):

We can use it to visualize both the input data and the weight vectors. Basically its main objective is to plots the input vectors as green dots and shows how the SOM classifies the input space by showing blue-gray dots for each neuron's weight vector and connecting neighboring neurons with red lines.

For example, See the below code:
load simplecluster_dataset

net = newsom(simpleclusterInputs, [10 10]);

net = train(net, simpleclusterInputs);

plotsompos(net, simpleclusterInputs);

Figure A.136: SOM weight positions using plotsompos.
References:

- MathWorks Inc., Image Processing Tool Box Users Guide
- MathWorks Inc., Neural Network Tool Box Users Guide