MATLAB Tutorial

Claudio Angione

Adapted from ETH Zurich, Department of Biosystems Science and Engineering (D-BSSE), and from MATLAB official tutorials

Contents

1 Introduction

1.1 What is MATLAB ?

MATLAB is a widely- used environment for technical computing with a focus on matrix operations. The name MATLAB stands for "MATrix LABoratory" and was originally designed as a tool for doing numerical computations with matrices and vectors. It has since grown into a high-performance language for technical computing. MATLAB integrates computation, visualization, and programming in an easy-to-use environment, and allows easy matrix manipulation, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs in other languages. Typical areas of use include:

- Math and Computation
- Modeling and Simulation
- Data Analysis and Visualization
- Application Development
- Graphical User Interface Development

1.2 Getting Started

Window Layout The first time you start MATLAB, the desktop appears with the default layout, as shown in Figure 1. The MATLAB desktop consists of the following parts:

- **Command Window:** Run MATLAB statements.
- **Current Directory:** To view, open, search for, and make changes to MATLAB related directories and files.
- **Command History:** Displays a log of the functions you have entered in the Command Window. You can copy them, execute them, and more.
- **Workspace:** Shows the name of each variable, its value, and the Min and Max entry if the variable is a matrix.

In case that the desktop does not appear with the default layout, you can change it from the menu Desktop → Desktop Layout → Default.

1.3 Editor

The MATLAB editor (Figure 2) can be used to create and edit M–files, in which you can write and save MATLAB programs. A M-file can take the form of a script file or a function. A script file contains a sequence of MATLAB statements; the statements contained in a script file can be run in the specified order, in the MATLAB command window simply by typing the name of the file at the command prompt. M–files are very useful when you use a sequence of commands over and over again, in many different MATLAB sessions and you do not want to manually type these commands at the command prompt every time you want to use them.

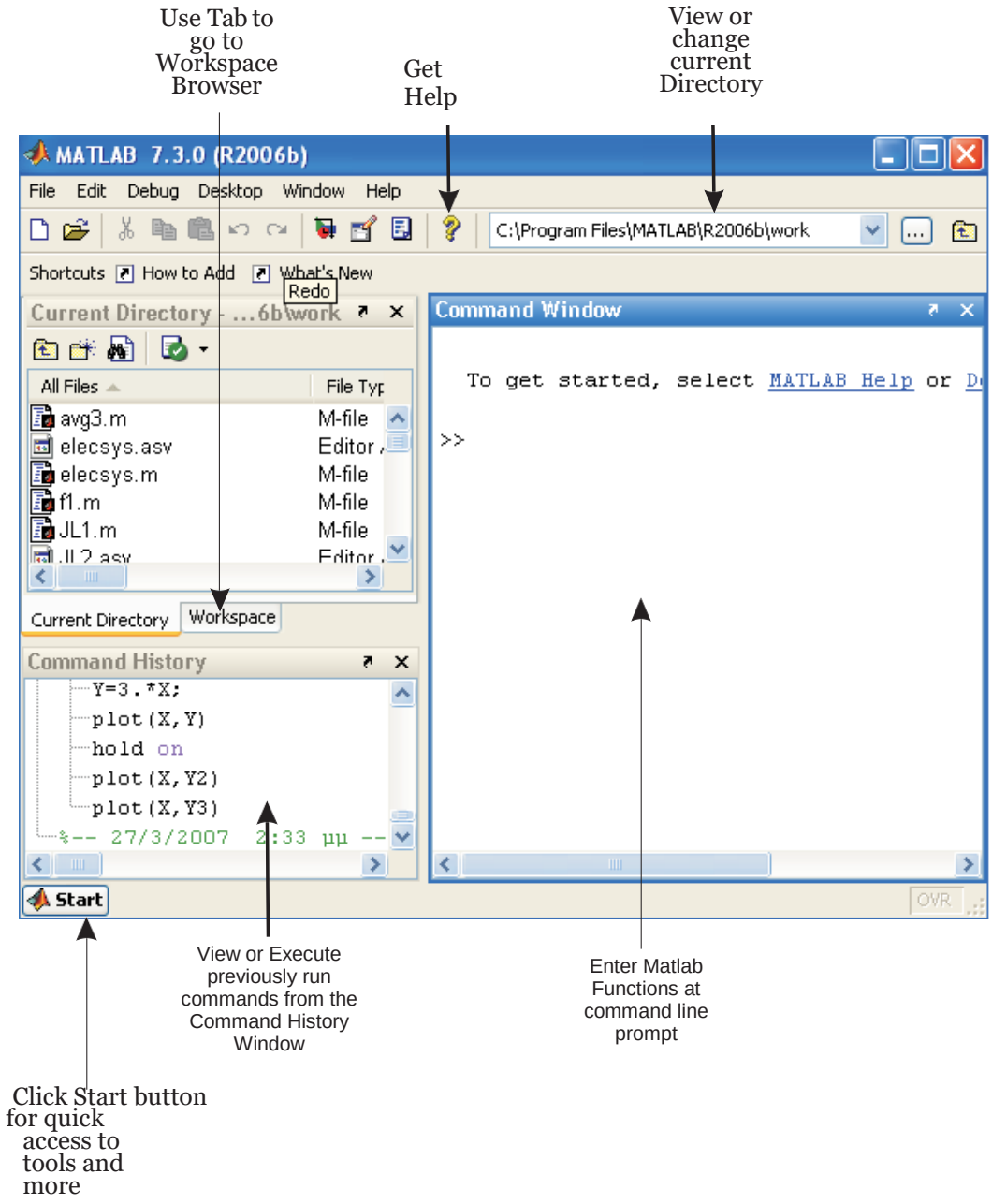


Figure 1: MATLAB Desktop (default layout)

You can run a script, or a function that does not require an input argument, directly from the Editor/Debugger either by pressing *F*5 or selecting Save File and Run from the Debug menu. If you only want to run a part of a script, you can use the mouse to highlight the corresponding lines in the M-file and press *F*9. The results are shown in Command Window.

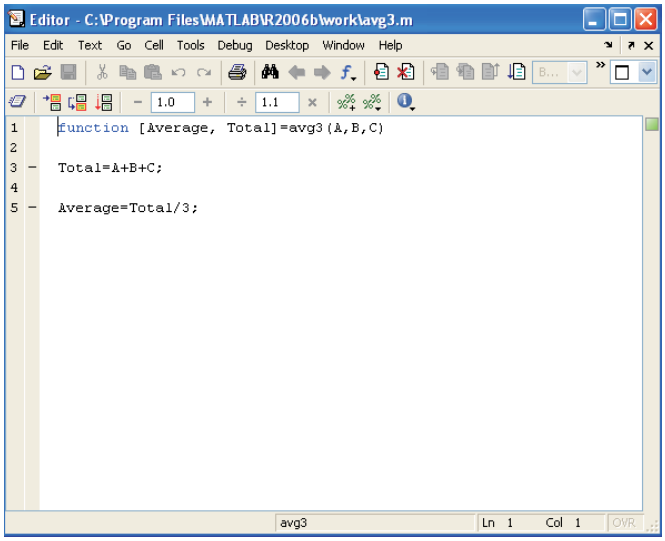


Figure 2: MATLAB Editor

1.4 MATLAB Help

MATLAB has an extensive built-in help system, which contains detailed documentation for all of the commands and functions of MATLAB . There are different ways to ask for help when using MATLAB : **Command Line**

- **HELP:** HELP FUN displays a description of and syntax for the function FUN in the Command Window (e.g., help plot).
- **DOC:** DOC FUN displays the help browser for the MATLAB function FUN (e.g. doc help). You can invoke the MATLAB help browser by typing "helpbrowser" at the MATLAB command prompt, clicking on the help button, or by selecting *Start MATLAB Help* from the MATLAB desktop.

1.5 Exercise

OK, now it's the time to play around a bit with MATLAB:

1. Start MATLAB.
2. Feel free to click around different segments in the MATLAB window, try resizing or closing some of them.
3. Now recover the desktop"default layout", so that your MATLAB window contains the main features shown in Figure 1 again.
4. Change the working directory to "Bioinformatics2010/your name". (Make the directories if needed!)
5. Try out the command window, the easiest way to use command window is to think of it simply as a normal calculator.
6. Done!

2 Basic data types and operations

2.1 Variable names

There are some specific rules for how you can name your variables, so you have to be careful.

- Only use primary alphabetic characters (i.e., "A-Z"), numbers, and the underscore character in your variable names.
- You cannot have any spaces in your variable names, so, for example, using "this is a variable" as a variable name is not allowed (in general, you can use the underscore character to replace space in your variable name).
- MATLAB is **case sensitive**. What this means for variables is that the same text, with different combinations of capital and small case letters, will not be interpreted the same in MATLAB . For example, "VaRIAbLe", "variable", "VARIABLE" and "variablE" would all be considered distinct variables in MATLAB .

2.2 Numerical variables and operations

Variables are defined in MATLAB by usage, i.e. there's no explicit declaration section needed. Numerical variables are by default defined as "double". The most basic operations with MATLAB are arithmetic operations. The basic arithmetic operators are +, -, *, /, ^(power). These operators can be used together with brackets (). As with all programming languages, special care should be given to how a mathematical expression is written. For example, $5 + 10/2 * 3 = 5 + (10/2) * 3 = 20$, which is not equal to $5 + 10/(2 * 3)$. In general, MATLAB follows the order:

1. Parentheses ( )
2. Exponentiation^
3. Multiplication and division, $*$, /, from left to right
4. Plus and minus, $+$, $-$, from left to right

For example:

$$\begin{aligned} 3 + 5/2 * 4 - 2\text{^}3 + (5 * 2) &= 3 + 5/2 * 4 - 2\text{^}3 + 10 \\ &= 3 + 5/2 * 4 - 8 + 10 \\ &= 3 + 2.5 * 4 - 8 + 10 \\ &= 3 + 10 - 8 + 10 \\ &= 15 \end{aligned}$$

MATLAB always stores the result of the latest computation in a variable named ans (short for "answer"). **Note:** the same *ans* variable will be rewritten at the next instruction, so if you need to save it you need to choose another name. To store variables for a longer time we can specify our own names:

```
>> x = 5+2^2

x =

     9
```

and then we can use these variables in computations:

```
>> y = 2*x

y =

    18
```

These are examples of **assignment statements**: values are assigned to variables. **Each variable must be assigned a value before it may be used on the right of an assignment statement**. If you do not want to see the result of a statement in the Command Window, which is typically the case for intermediate statements, we can terminate the line with a semi–colon:

```
>> x = 5+2^2;
>> y = 2*x;
>> z = x^2+y^2

z =

   405
```

Notes:

- Other types of numerical variables can be defined explicitly if needed as:

  `char, single, int8, int16, int64, uint8, uint16, uint64.`

- MATLAB is accurate but does not do exact arithmetic!

  ```
  >> 3*(5/3 - 4/3) - 1
  ans =
        4.440892098500626e-16
  ```

- MATLAB identifies *i, j, pi* also as numbers and to some extent *Inf* and *NaN* :

  ```
  >> pi
  ans = 3.1416

  >> (-1)^(0.5)
  ans = 0.0000 + 1.0000i

  >> log(0)
  ans = -Inf

  >> 0/0
  ans = NaN
  ```

2.3 Logical variables and operators

A logical variable can be ”true” or ”false”, or one and zero in binary system.

```
>>S = 2>4
S =     0
```

Boolean algebra can be done in MATLAB using the logical operators AND (&) , OR (|) and NOT (~). For example given an arbitrary Boolean variable S:

```
>> S | ~S
ans =     1
```

Note: MATLAB implicitly "casts" data types to avoid syntax errors:

- Logical values are converted to 1 and 0 when used as numbers.

```
>> (2 < 3) * 3
ans =     3
>> true * 3
ans =     3
```

- All numbers hold "true" Boolean value when used in logical expressions, except for 0 itself which is "false".

```
>> false

ans =   1
```

2.4 String variables

You can also assign strings of text to variables, not just numbers, with single quotes (not double quotes) around the text. For example:

```
>>  w  = 'Goodmorning';
>> w
w =
Goodmorning
```

2.5 Vectors and matrices

Matrices are not a type of data but they are n-dimensional arrays of basic MATLAB data-types. MATLAB treats all variables equally as matrices. Traditional matrices and vectors are two- and one-dimensional cases of these structures, respectively, and scalar numbers are simply 1-by-1 matrices. A matrix is defined inside a pair of square brackets ([]). The punctuation marks comma (,), and semicolon (;) are used as a row separator and column separator, respectively. The examples below illustrate how vectors and matrices can be created in MATLAB .

```
>> A = [1, 2, 3]
A =
     1     2     3
>> A = [1; 2; 3]
A =
     1
     2
     3
>> A = [1 2 3; 4 5 6]
A =

     1     2     3
     4     5     6
```

Simple matrices can also be created using functions such as: ones, zeros, eye, diag. For example:

```
>> A = eye(3)
A =
     1     0     0
     0     1     0
     0     0     1
>> A = diag([1 2, 3])
A =
     1     0     0
     0     2     0
     0     0     3
```

Concatenating matrices are straightforward MATLAB as long as their dimensions are consistent. For example:

```
>> A = [1, 2, 3];B = [4, 5, 6];
>> C = [A, B]
C =
     1     2     3     4     5     6
>> C = [A; B]
C =
     1     2     3
     4     5     6
```

Colon operator

The colon operator allows you to create vectors with a sequence of values from the *start value* to the *stop value* with a specified *increment* value. The increment value can also be negative or non-integer. For example:

```
>> A = [-2: 2]
A =
    -2    -1     0     1     2
>> A = [2: -1: -2]
A =
     2     1     0    -1    -2
>> A = [0.5: 1.5: 6]
A =
    0.5000                3.500      5.000
```

Vector and matrix indexing

Once a vector or a matrix is created you might needed to access only a subset of the data. This can be done with indexing. Each element of a matrix is indexed with the row and column of the element. The entry in the *i*th row and *j*th column is denoted mathematically by A_{ij} and in MATLAB by $A(i, j)$. Consider the matrix:

```
>> C = [1 2 3;4 5 6;7 8 9;10 11 12]
C =
     1     2     3
     4     5     6
```

```
     7     8     9
    10    11    12
```

the element on the 3rd row of the 2nd column can be accessed like:

```
>> C(3, 2)
ans =     8
```

The colon operator can also be used to access the whole or a set consecutive elements within a dimension of a matrix. Given the data matrix *C* from above we have:

```
>> C(:, 3)
ans =
     3
     6
     9
    12
>> C(2:3, 2:3)
ans =
     5     6
     8     9
```

Matrix operations

Basic matrix operations are straightforward in MATLAB :

- Addition: >> C = A+B;
- Subtraction: >> C = A-B;
- Transposing: >> C = A';
- Matrix multiplication: >> C = A*B;
- Element-wise multiplication: >> C = A .*B;
- Exponentiation: >> C = A^p; (where p is a scalar.)
- Element-wise exponentiation: >> C = A.^p; (where p is a scalar.)

Note: Matrices must have compatible dimensions!

2.6 Exercise

1. Create a row vector **v** with values (1, 2, 3, 5, 11, 7, 13).
2. Change the value of the 5*th* and the 6*th* element of the **v** to 7 and 11 respectively. Try doing this with only one command as well.
3. Create a vector *Five5* out of all multiples of 5 that fall between 34 and 637. By the way, how many are they? (Tip: look up the command "length" or "numel" in help.)
4. Double click on the variable *Five5* in the *Workspace* to inspect your results in the *Variable Editor*.
5. Can you use the colon operator to extract the numbers in the vector *Five5* that are multiples of 5 only but not 10?
6. Why v*v and v.*v have different size? What do they represent?
7. What do v^2 and v.^2 represent?

3 Plotting basics

3.1 The plot command

The most basic plotting command in MATLAB is plot. The plot function has different forms, depending on the input arguments. If **y** is a vector, plot(y) produces a piecewise linear graph of the elements of **y** versus the index of the elements of **y**. If you specify two vectors as arguments, plot(x,y) produces a graph of **y** versus **x**, i.e. the points $((x_1,y_1),(x_2,y_2),(x_3,y_3),\ldots)$, are connected with lines.

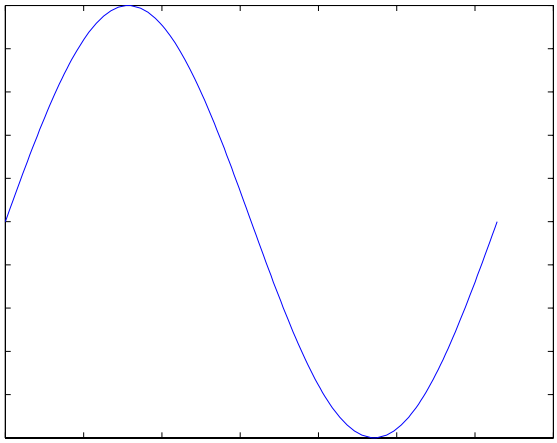


Figure 3: Plot of the sine function

For example, these statements use the colon operator to create a vector of **x** values ranging from 0 to 2π , compute the sine of these values, and plot the result:

```
>> x = 0:pi/100:2*pi;
>> y = sin(x);
>> plot(x,y)
```

The result is show in figure 3.
However, it is not always desirable to have the consecutive points to be connected to each other. In these cases the plot command can be used to draw scatter plots by other symbols instead of lines, for example with dots: plot(x, y, '.'). In general it is possible to specify color, line styles, and markers (such as plus signs or circles) when you plot your data using the plot command: plot(x,y,'color_style_marker') where *color style marker* is a string containing from one to four characters (enclosed in single quotation marks) constructed from a color, a line style, and a marker type. The strings are composed of combinations of the elements shown in figure 4.
You can also edit color, line style, and markers interactively. See MATLAB help of "Editing Plots" for more information.

Type	Values	Meanings
Color	'c' 'm' 'y' 'r' 'g' 'b' 'w' 'k'	cyan magenta yellow red green blue white black
Line style	'-' '--' ':' '-.' no character	solid dashed dotted dash-dot no line
Marker type	'+' 'o' '*' 'x' 's' 'd' '^' 'v' '>' '<' 'p' 'h' no character or none	plus mark unfilled circle asterisk letter x filled square filled diamond filled upward triangle filled downward triangle filled right-pointing triangle filled left-pointing triangle filled pentagram filled hexagram no marker

Figure 4: The plot command: Line styles and colours.

3.2 The hist command

Histogram plots are one of the other basic types of plots in MATLAB and can be produced using the hist function. The easiest form of using this function is hist(x), in which **x** denotes a vector. The command divides the range of the values in **x** to ten bins and plots the distribution of the element counts in each bin using bar plots.

An additional scalar parameter can be added afterwards in order to tune the number of bins, for example the following commands produce 10,000 normally distributed random numbers, plot their histogram using 100 bins:

```
>> x = randn(10000,1);
>> hist(x, 100)
```

The result is shown in Figure 5.

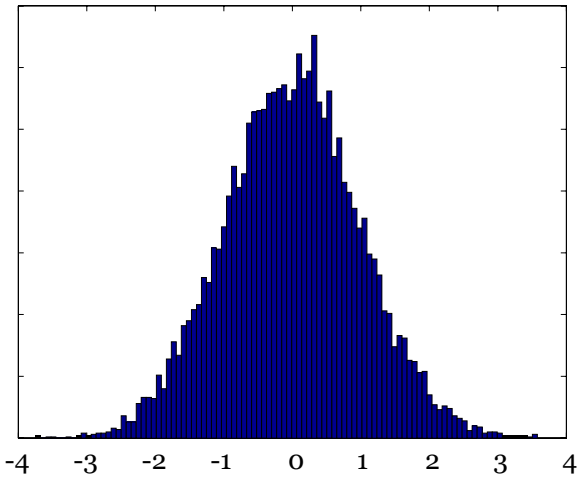


Figure 5: Histogram plot of 10,000 normally distributed data.

3.3 Multiple Plots and Subplots

To create new figures hosting new plots, use the "figure" command.

You may also want to create more than one plot in the same figure. This can be done using the hold command. Normally when you use the plot command any previous plot in the figure is erased, and replaced by the new plot. However, if you first type "hold on" in the command prompt, all new plots will be superimposed in the same figure. The command "hold off" tells MATLAB to return to the default mode (i.e., each new plot will replace the previous plot in the figure). For example:

```
>> X = [1 3 4 6 8 12 18];
>> Y1 = 3*X;
>> Y2 = 4*X+5;
>> Y3 = 2*X-3;
>>  plot(X,Y1);
>> hold on
>>  plot(X,Y2);
>>  plot(X,Y3);
```

You can also create multiple plots in the same figure, but with each of them in a separate subfigure (i.e., each with their own axes). You can do this with the **subplot** command. If you type subplot(M,N,P) at the command prompt, MATLAB will divide the window into a bunch of subfigures. There will be M rows and N columns of subfigures and MATLAB will place the result of the next "plot" command in the P*th* subfigure (where the first subfigure is in the upper left). For example:

```
>>  subplot(3,1,1)
>> plot(X,Y1)
>>  subplot(3,1,2)
>> plot(X,Y2)
>>  subplot(3,1,3)
>> plot(X,Y3)
```

3.4 Exercise

1. Initialize the random seed with command:
   `>> S= RandStream('mrg32k3a');`
2. Create a random vector **u** of 1,000 uniformly distributed numbers. (Tip: use the *rand* command.)
3. Create a random vector **n** of 1, 000 normally distributed numbers. (Tip: use the *randn* command.)
4. Create a new plot window using the command *Figure*.
5. Plot the distributions and scatter plot of the elements in **u** and **n** in the figure window. (Three subplots in total.)
6. Make yourself familiar with the interactive plot editor, accessible from "View Property Editor" in the figure window menu bar. Give fancy labels to your axes and figures, so that they look scientific enough!
7. Save your figures with ".eps" extension in your folder, and close the figure.
8. Save your workspace to a file called "My Workspace" in your folder using *save* command.
9. Close MATLAB , done!

4 MATLAB functions

MATLAB has an extremely wide range of built-in functions. Additionally, it is also simple to define your own functions in MATLAB as any other programming languages.

4.1 Simple Built-in Functions

Here are some basic instances of the ready MATLAB functions:

- Trigonometric functions: sin, cos, tan, asin, acos, atan (their arguments should be in radians).
- Exponential: exp, log, log2, log10
- Random number generator: rand, randn, randperm, mvnrnd
- Data analysis: min, max, mean, median, std, var, cov
- Linear algebra: norm, det, rank, inv, eig, svd, null, pinv
- Strings: strcmp, strcat, strfind, sprintf, sscanf, eval
- Files: save, load, csvwrite, csvread, fopen, fprintf, fscanf
- Other: abs, sign, sum, prod, sqrt, floor, ceil, round, sort, find
- Master key: help, doc, and the external function Google!

4.2 Exercise

1. Open MATLAB , and load the data file "My Workspace" from the previous section into MATLAB. (Tip: you can use the "load" button in the *Workspace* or the *load* command.)
2. Can you find how many elements in **n** are larger than 0.5 ?
3. What is the index of the smallest element in in the vector **n** ? (Tip: check out the *min* function.)
4. Try calculating the mean and standard deviation of n. What values should we expect? Whay they are slightly different from what we expected?
5. How can we make a new vector called 'n1' that follows a normal distribution with expected mean 5 and expected standard deviation 2?
6. Compute *meanU* as the average of the vector **u**. (Tip: use *mean* function.)
7. Find the location of the elements in **u** that are larger than 0.5 using the command
   `>> I = find(u>0.5).`
8. Set all of the elements in **u** with larger than 0.5 to zero. How much does the mean of **u** shrink ?

4.3 MATLAB scripting

To avoid writing the same code over and over again you can create so called M-files (they have the file extension .m). These can be either scripts or functions. The main differences are:

- Scripts work like an extended command line statement. All variables are shared between the workspace and the script. Scripts have no input or output arguments.
- Functions have their own workspace. They are usually called with one or several input arguments and return one or several output arguments. They are declared by the keyword function.

Creating MATLAB scripts is easy. Just call the MATLAB editor:

```
>> edit example_script.m
```

Now write your lines of code and save the file somewhere in the MATLAB search path (e.g. the current working directory) and call it from the command line by:

```
>> example_script
```

The script will be executed line-by-line.

4.4 Defining a function

You can easily extend the functionality of MATLAB by creating new functions. The general form of a function declaration is:

```
function [return_args] = function_name(input_args)
function body ...
```

Thus, a MATLAB function has a name and optionally input and return arguments. The return arguments must be assigned somewhere in the function body.
Example: the following function computes the product of two numbers a and b.

```
function p = product(a,b)

p = a*b;
```

Saved under the file name product.m it can be called from the command line with the "product" command.

```
>> x = product(3,4)

x =
      12
```

Example: this function sets to zero all values in the input vector which are below a given threshold. It returns the altered vector and also the indices of the replaced values.

```
function [v,idx] = findBelowThresh(v,thresh)
idx = find(v<thresh);
v(idx) = 0;
```

As a matter of convention, you should always call the M-file and the function by the same name. However, make sure that it does not conflict with a name that is already taken by another function.

4.5 Exercises

1. Create a function that returns the average of given input vector.
2. Create a function that computes the binomial coefficient $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
3. Create a function that plots $\sin(\omega x)$ for a given input frequency ω in the range $0 \le x \le 2\pi$.
4. Create a function to plot the surface $\sin(\omega_1 x) * \cos(\omega_2 y)$ in the range $0 \le x \le 2\pi$, $0 \le y \le 2\pi$. You can use the meshgrid function to create the x and y matrices.

5 Flow control

5.1 Checking conditions: if, else, elseif

Conditional statements check a given expression and based on the outcome execute certain parts of the code. You can also check several conditions in a sequential order with `elseif` statements.

```
if expression
   statements
elseif expression
     statements
elseif expression
     statements
...
else
      statements
end
```

Example: the following function computes the factorial *n*! by recursive function calls.

```
function f = fac(n)
if n == 0
   f = 1;
else
   f = n*fac(n-1);
end
```

5.2 Iterations: for and while loops

Iterations are defined by for and while loops. The difference between both is that for-loops have a fixed number of cycles. While-loops stop until a certain condition is matched. The for loop has the general syntax:

```
for variable = vector
     statements
end
```

Example: the following function computes the Fibonacci series.

```
function f = fibonacci(n)
f = zeros(n,1);
f(1:2) = [1 1];
for l=3:n
   f(l) = f(l-2) + f(l-1);
end
```

The while loop format is

```
while condition statements
     do stuff
end
```

Example: the following function sorts the elements of a vector in ascending order.

```
function v = gsort(v) pos  = 2;
while  pos  <=  length(v)
   if v(pos)>v(pos-1)
      % move to the  next  position
      pos  =  pos  + 1;
   else % swap the values
        foo = v(pos-1);
        v(pos-1) = v(pos);
        v(pos)  =  foo;
        if pos > 2 % decrease position
             pos  =  pos  - 1;
        end
   end
end
```

For and while loops can be interrupted with the break statement. Generally, for and while loops should be avoided in MATLAB as they are very slow. Instead you should try to use built-in MATLAB functions or vectorize the code with the dot-operator.

5.3 Exercises

1. Extend the factorial function defined above, such that it checks for a positive input argument.
2. Write a function that calculates a random walk trajectory based on the iteration $x_{i+1} = x_i + E$, where E is a uniformly distributed random variable between -1 and 1. Plot the resulting trajectory.
3. Sort the elements in descending order.
4. Try plotting different random walks on the same plot, always starting from the same initial point.

6 Working with GEO Series Gene Expression Data

This example shows how to retrieve gene expression data series (GSE) from the NCBI Gene Expression Omnibus (GEO) and perform basic analysis on the expression profiles.

6.1 Introduction

The NCBI Gene Expression Omnibus (GEO) is the largest public repository of high-throughput microarray experimental data. GEO data have four entity types including GEO Platform (GPL), GEO Sample (GSM), GEO Series (GSE) and curated GEO DataSet (GDS).

A Platform record describes the list of elements on the array in the experiment (e.g., cDNAs, oligonucleotide probesets). Each Platform record is assigned a unique and stable GEO accession number (GPLxxx).

A Sample record describes the conditions under which an individual Sample was handled, the manipulations it underwent, and the abundance measurement of each element derived from it. Each Sample record is assigned a unique and stable GEO accession number (GSMxxx).

A Series record defines a group of related Samples and provides a focal point and description of the whole study. Series records may also contain tables describing extracted data, summary conclusions, or analyses. Each Series record is assigned a unique GEO accession number (GSExxx).

A DataSet record (GDSxxx) represents a curated collection of biologically and statistically comparable GEO Samples. GEO DataSets (GDSxxx) are curated sets of GEO Sample data.

More information about GEO can be found in GEO Overview. Bioinformatics Toolbox™ provides functions that can retrieve and parse GEO format data files. GSE, GSM, GSD and GPL data can be retrieved by using the getgeodata function. The getgeodata function can also save the retrieved data in a text file. GEO Series records are available in SOFT format files and in tab-delimited text format files. The function geoseriesread reads the GEO Series text format file. The geosoftread function reads the usually quite large SOFT format files.

In this example, you will retrieve the GSE5847 data set from GEO database, and perform statistical testing on the data. GEO Series GSE5847 contains experimental data from a gene expression study of tumor stroma and epithelium cells from 15 inflammatory breast cancer (IBC) cases and 35 non-inflammatory breast cancer cases (Boersma et al. 2008).

6.2 Retrieving GEO Series Data

The function getgeodata returns a structure containing data retrieved from the GEO database. You can also save the returned data in its original format to your local file system for use in subsequent MATLAB® sessions. Note that data in public repositories is frequently curated and updated; therefore the results of this example might be slightly different when you use up-to-date datasets.

```
gseData = getgeodata('GSE5847', 'ToFile', 'GSE5847.txt')
```

Use the geoseriesread function to parse the previously downloaded GSE text format file.

```
gseData = geoseriesread('GSE5847.txt')
gseData =

  struct with fields:
```

```
    Header: [1×1 struct]
      Data: [22283×95 bioma.data.DataMatrix]
```

The structure returned contains a Header field that stores the metadata of the Series data, and a Data field that stores the Series matrix data.

6.3 Exploring GSE Data

The GSE5847 matrix data in the Data field are stored as a DataMatrix object. A DataMatrix object is a data structure like MATLAB 2-D array, but with additional metadata of row names and column names. The properties of a DataMatrix can be accessed like other MATLAB objects.

```
get(gseData.Data)
          Name: ''
      RowNames: {22283×1 cell}
      ColNames: {1×95 cell}
         NRows: 22283
         NCols: 95
         NDims: 2
   ElementClass: 'double'
```

The row names are the identifiers of the probe sets on the array; the column names are the GEO Sample accession numbers.

```
gseData.Data(1:5, 1:5)
ans =

            GSM136326    GSM136327    GSM136328    GSM136329    GSM136330
   1007_s_at    10.45       9.3995       9.4248       9.4729       9.2788
   1053_at     5.7195       4.8493       4.7321       4.7289       5.3264
   117_at      5.9387       6.0833        6.448       6.1769       6.5446
   121_at      8.0231       7.8947        8.345       8.1632       8.2338
   1255_g_at   3.9548       3.9632       3.9641       4.0878       3.9989
```

The Series metadata are stored in the Header field. The structure contains Series information in the Header.Series field, and sample information in theHeader.Sample field.

```
gseData.Header
ans =

  struct with fields:

     Series: [1×1 struct]
    Samples: [1×1 struct]
```

The Series field contains the title of the experiment and the microarray GEO Platform ID.

```
gseData.Header.Series
ans =

  struct with fields:

                title: 'Tumor and stroma from breast by LCM'
        geo_accession: 'GSE5847'
               status: 'Public on Sep 30 2007'
      submission_date: 'Sep 15 2006'
```

```
          last_update_date: 'Nov 14 2012'
                pubmed_id: '17999412'
                  summary: 'Tumor epithelium and surrounding stromal ...'
           overall_design: 'We applied LCM to obtain samples enriched...'
                     type: 'Expression profiling by array'
              contributor: 'Stefan,,Ambs...'
                sample_id: 'GSM136326 GSM136327 GSM136328 GSM136329 G...'
             contact_name: 'Stefan,,Ambs'
       contact_laboratory: 'LHC'
        contact_institute: 'NCI'
          contact_address: '37 Convent Dr Bldg 37 Room 3050'
             contact_city: 'Bethesda'
            contact_state: 'MD'
  contact_zip0x2Fpostal_code: '20892'
          contact_country: 'USA'
       supplementary_file: 'ftp://ftp.ncbi.nlm.nih.gov/pub/geo/DATA/s...'
              platform_id: 'GPL96'
            platform_taxid: '9606'
              sample_taxid: '9606'
                  relation: 'BioProject: http://www.ncbi.nlm.nih.gov/b...'

gseData.Header.Samples
ans =

  struct with fields:

                     title: {1×95 cell}
              geo_accession: {1×95 cell}
                    status: {1×95 cell}
           submission_date: {1×95 cell}
          last_update_date: {1×95 cell}
                      type: {1×95 cell}
             channel_count: {1×95 cell}
           source_name_ch1: {1×95 cell}
            organism_ch1: {1×95 cell}
       characteristics_ch1: {2×95 cell}
            molecule_ch1: {1×95 cell}
      extract_protocol_ch1: {1×95 cell}
             label_ch1: {1×95 cell}
       label_protocol_ch1: {1×95 cell}
             taxid_ch1: {1×95 cell}
           hyb_protocol: {1×95 cell}
          scan_protocol: {1×95 cell}
           description: {1×95 cell}
         data_processing: {1×95 cell}
            platform_id: {1×95 cell}
           contact_name: {1×95 cell}
       contact_laboratory: {1×95 cell}
        contact_institute: {1×95 cell}
          contact_address: {1×95 cell}
             contact_city: {1×95 cell}
            contact_state: {1×95 cell}
  contact_zip0x2Fpostal_code: {1×95 cell}
          contact_country: {1×95 cell}
       supplementary_file: {1×95 cell}
          data_row_count: {1×95 cell}
```

The data_processing field contains the information of the preprocessing methods, in this case the Robust Multi-array Average (RMA) procedure.

```
gseData.Header.Samples.data_processing(1)
ans =

  cell

    'RMA'
```

The source_name_ch1 field contains the sample source:

```
sampleSources = unique(gseData.Header.Samples.source_name_ch1);
sampleSources{:}
ans =

human breast cancer stroma


ans =

human breast cancer tumor epithelium
```

The field Header.Samples.characteristics_ch1 contains the characteristics of the samples.

```
gseData.Header.Samples.characteristics_ch1(:,1)
ans =

  2×1 cell array

    'IBC'
    'Deceased'
```

Determine the IBC and non-IBC labels for the samples from the Header.Samples.characteristics_ch1 field as group labels.

```
sampleGrp = gseData.Header.Samples.characteristics_ch1(1,:);
```

6.4 Retrieving GEO Platform (GPL) Data

The Series metadata told us the array platform id: GPL96, which is an Affymetrix® GeneChip® Human Genome U133 array set HG-U133A. Retrieve the GPL96 SOFT format file from GEO using the getgeodata function. For example, assume you used the getgeodata function to retrieve the GPL96 Platform file and saved it to a file, such as GPL96.txt. Use the geosoftread function to parse this SOFT format file.

```
gplData = geosoftread('GPL96.txt')
gplData =

  struct with fields:

          Scope: 'PLATFORM'
      Accession: 'GPL96'
         Header: [1×1 struct]
```

```
    ColumnDescriptions: {16×1 cell}
         ColumnNames: {16×1 cell}
                Data: {22283×16 cell}
```

The ColumnNames field of the gplData structure contains names of the columns for the data:

```
gplData.ColumnNames
ans =

  16×1 cell array

    'ID'
    'GB_ACC'
    'SPOT_ID'
    'Species Scientific Name'
    'Annotation Date'
    'Sequence Type'
    'Sequence Source'
    'Target Description'
    'Representative Public ID'
    'Gene Title'
    'Gene Symbol'
    'ENTREZ_GENE_ID'
    'RefSeq Transcript ID'
    'Gene Ontology Biological Process'
    'Gene Ontology Cellular Component'
    'Gene Ontology Molecular Function'
```

You can get the probe set ids and gene symbols for the probe sets of platform GPL69.

```
gplProbesetIDs = gplData.Data(:, strcmp(gplData.ColumnNames, 'ID'));
geneSymbols = gplData.Data(:, strcmp(gplData.ColumnNames, 'Gene Symbol'));
```

Use gene symbols to label the genes in the DataMatrix object gseData.Data. Be aware that the probe set IDs from the platform file may not be in the same order as in gseData.Data. In this example they are in the same order.

Change the row names of the expression data to gene symbols.

```
gseData.Data = rownames(gseData.Data, ':', geneSymbols);
```

Display the first five rows and five columns of the expression data with row names as gene symbols.

```
gseData.Data(1:5, 1:5)
ans =

          GSM136326    GSM136327    GSM136328    GSM136329    GSM136330
  DDR1      10.45       9.3995       9.4248       9.4729       9.2788
  RFC2      5.7195       4.8493       4.7321       4.7289       5.3264
  HSPA6     5.9387       6.0833        6.448       6.1769       6.5446
  PAX8      8.0231       7.8947        8.345       8.1632       8.2338
  GUCA1A    3.9548       3.9632       3.9641       4.0878       3.9989
```

6.5 Analyzing the Data

Bioinformatics Toolbox and Statistics and Machine Learning Toolbox™ offer a wide spectrum of analysis and visualization tools for microarray data analysis. However, because it is not our main goal

to explain the analysis methods in this example, you will apply only a few of the functions to the gene expression profile from stromal cells. For more elaborate examples about the gene expression analysis and feature selection tools available, see Exploring Gene Expression Data and Selecting Features for Classifying High-dimensional Data.

The experiment was performed on IBC and non-IBC samples derived from stromal cells and epithelial cells. In this example, you will work with the gene expression profile from stromal cells. Determine the sample indices for the stromal cell type from the gseData.Header.Samples.source_name_ch1 field.

```
stromaIdx = strcmpi(sampleSources{1}, gseData.Header.Samples.source_name_ch1);
```
Determine number of samples from stromal cells.

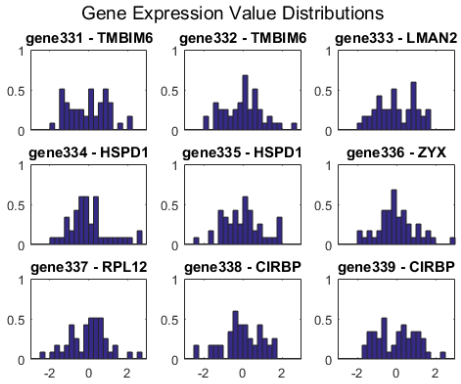
```
nStroma = sum(stromaIdx)
nStroma =

    47

stromaData = gseData.Data(:, stromaIdx);
stromaGrp = sampleGrp(stromaIdx);
```
Determine the number of IBC and non-IBC stromal cell samples.

```
nStromaIBC = sum(strcmp('IBC', stromaGrp))
nStromaIBC =

    13

nStromaNonIBC = sum(strcmp('non-IBC', stromaGrp))
nStromaNonIBC =

    34
```

You can also label the columns in stromaData with the group labels:

```
stromaData = colnames(stromaData, ':', stromaGrp);
```
Display the histogram of the normalized gene expression measurements of a specified gene. The x-axes represent the normalized expression level. For example, inspect the distribution of the gene expression values of these genes.

```
fID = 331:339;
zValues = zscore(stromaData.(':')(':'), 0, 2);
bw = 0.25;
edges = -10:bw:10;
bins = edges(1:end-1) + diff(edges)/2;
histStroma = histc(zValues(fID, :)', edges) ./ (stromaData.NCols*bw);
figure;
for i = 1:length(fID)
   subplot(3,3,i);
   bar(edges, histStroma(:,i), 'histc')
   xlim([-3 3])
   if i <= length(fID)-3
      ax = gca;
      ax.XTickLabel = [];
   end
   title(sprintf('gene%d - %s', fID(i), stromaData.RowNames{fID(i)}))
end
suptitle('Gene Expression Value Distributions')
```



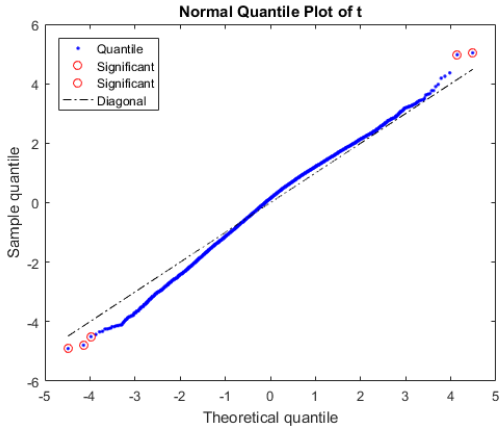
The gene expression profile was accessed using the Affymetrix GeneChip more then 22,000 features on a small number of samples (~100). Among the 47 tumor stromal samples, there are 13 IBC and 34 non-IBC. But not all the genes are differentially expressed between IBC and non-IBC tumors. Statistical tests are needed to identify a gene expression signature that distinguish IBC from non-IBC stromal samples.

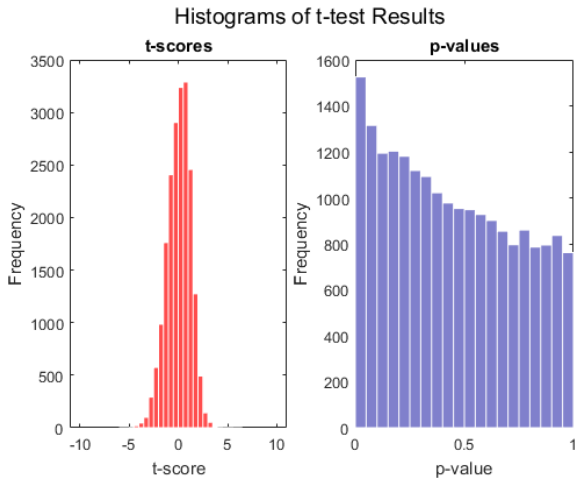
Use genevarfilter to filter out genes with a small variance across samples.

```
[mask, stromaData] = genevarfilter(stromaData);
stromaData.NRows
ans =

    20055
```

Apply a t-statistic on each gene and compare *p-values* for each gene to find significantly differentially expressed genes between IBC and non-IBC groups by permuting the samples (1,000 times for this example).

```
rng default
[pvalues, tscores] = mattest(stromaData(:, 'IBC'), stromaData(:, 'non-IBC'),...
                    'Showhist', true', 'showplot', true, 'permute', 1000);
```





Select the genes at a specified p-value.

```
sum(pvalues < 0.001)
ans =

   52
```

There are about 50 genes selected directly at *p-values* < 0.001.

Sort and list the top 20 genes:

```
testResults = [pvalues, tscores];
testResults = sortrows(testResults);
testResults(1:20, :)
ans =

                p-values     t-scores
  INPP5E          2.3337e-05     5.0389
  ARFRP1 /// IGLJ3    2.7575e-05     4.9753
  USP46           3.4346e-05    -4.9054
  GOLGB1          4.7706e-05    -4.7928
  TTC3            0.00010702    -4.5053
  THUMPD1         0.00013185    -4.4317
                  0.00016066     4.3656
  MAGED2          0.00017047    -4.3444
  DNAJB9          0.00017833    -4.3266
  KIF1C           0.00022129     4.2504
                  0.00022251    -4.2482
  DZIP3           0.00022425    -4.2454
  COPB1           0.00023203    -4.2332
  PSD3            0.00024659    -4.2138
  PLEKHA4         0.00026506     4.186
  DNAJB9          0.00027687    -4.1708
  CNPY2           0.00028039    -4.1672
  USP9X           0.00028454    -4.1619
  SEC22B          0.00030187    -4.1392
  GFER            0.00030527    -4.1352
```