A Whirlwind Tour of MATLAB

for Students of MACM 316

John M. Stockie

Department of Mathematics
Simon Fraser University
Burnaby, BC, Canada, V5A 1S6

stockie@math.sfu.ca

August 24, 2018
Version 1.6

Contents

1 Introduction 2

2 MATLAB fundamentals 2

3 Basic computations 3

4 Vectors 5

5 Matrices and linear algebra 8

6 Other built-in functions: 10

7 Loops 10

8 Command history and editing 11

9 Saving your commands in “script m-files” 11

10 Inline functions 12

11 Function m-files 13

12 Plotting 14

Copyright ©2008, 2012, 2013, 2018 by John M. Stockie (Simon Fraser University). This document may be freely distributed provided that it is not altered in any way and that credit is given to the author.

1 Introduction

This document is intended as a very brief introduction to the software package MATLAB, for students in the course MACM 316. It contains only just enough to get you started on the assignments, and so I would highly encourage you to explore MATLAB's capabilities on your own.

Many other MATLAB tutorials, guides and manuals are available, but (IMHO) most tend to be long-winded and intimidating for the student who requires a knowledge of only a small subset of the language's commands. My aim here is to give you the basics in 20 pages or less. For a more comprehensive treatment, please refer to other books and tutorials . . . a short list is included in the Bibliography. In particular, you may find the online tutorials [2], [3] and [7] quite helpful.

Throughout this document, certain standard conventions are used. MATLAB commands, which are intended to be typed at the MATLAB prompt, are displayed in a `blue typewriter font`. Corresponding output from MATLAB is shown in a `black typewriter font`.

2 MATLAB fundamentals

"MATLAB" stands for "Matrix Laboratory." It is an interactive software program for performing numerical computations. MATLAB was initially designed by Cleve Moler in the 1970s for use as a teaching tool, but it has since become a very successful commercial package used extensively in both education and industry. One of MATLAB's best features, from the point of view of the computational scientist, is its large built-in library of numerical routines and graphical visualisation tools.

Running MATLAB: To start MATLAB, you simply type the command `matlab` at your prompt, or if you are using Windows on the SFU Network, then click on the "MATLAB" icon. You will be presented with a text window that contains the MATLAB prompt:

```
>> _
```

This is the signal that MATLAB is waiting for you to type a command.

Directories and files: MATLAB can only access files that are in its working path or in the "currect working directory." My suggestion is that you save all of your files in a directory such as `f:\macm316\matlab\`.[1] Every time you start up MATLAB, it is a good idea to switch into this directory using the command:

```
>> cd 'f:\macm316\matlab\'
```

The single quotes are important!

> ***Note:*** *When presented with MATLAB commands in blue, you should type in the command* **not including** *the prompt, "*`>>`*".*

After executing the above command, you are ensured that you will have access to all of the files in this directory, and that all of your output and plot files will be saved to the same directory. Two other useful commands are `pwd`, which prints the current working directory, and `dir`, which gives a listing of your files.

An alternate (and more permanent) way to allow MATLAB access to your files, particularly if they reside in more than one directory, is to modify MATLAB's "search path," which is a list of directories to search for files. You can change the search path either using the `path` command or, if you are working on a Windows machine, by selecting the '`Options -> Change Path`' menu item.

Getting help: The most useful command in MATLAB is `help`. Use it liberally and use it often. For example, typing `help plot` will display documentation on the plotting command. You will often find that the text window

[1]The drive and/or exact location of the course directory may differ depending on which machine or system you are working on. The naming convention I use here is consistent with the MS Windows operating system. If a directory doesn't already exist, then create one for yourself.

is not large enough to fit all of the output from a MATLAB command. To allow you to page through the information one screen at a time, you can use **`more on`**, and then progress through pages of output by hitting the space bar. Typing **`more off`** will return to the default behaviour of no paging.

It's often the case that you don't know the exact name of a MATLAB command, in which case you can use **`lookfor`** to help you out. For example, typing **`lookfor logarithm`** will list all known MATLAB functions that have something to do with logarithms.

Saving your session: When working on assignments, it is very helpful to be able to save to a file all of the input and output from your current MATLAB session for later printing. The command **`diary`** is very useful for this purpose. At the prompt, you simply type:

```
>> diary( 'diary_file_name' )
>>       ... type your commands here ...
>> diary off
```

which will save all input and output between the **`diary`** commands to a file with the given name.

3 Basic computations

Arithmetic operations: MATLAB's arithmetic operations follow a syntax that is very similar to that in other languages that you might be familiar with such as C, C++, Fortran, Java, etc. The assignment operator is **`=`**, the basic arithmetic operators are **`+`**, **`-`**, **`*`**, **`/`**, and the exponentiation operator is **`^`**. Using these basic operators, MATLAB can be used as simple calculator:

```
>> 10 - 13 + 4 * 2 / 5
ans =
   -1.4000
>> 2^5 - 1
ans =
   31
```

Matlab returns the result of the most recent computation as **`ans = ...`**. The precedence of operators is the usual order – exponentiation, followed by multiplication/division, followed by addition/subtraction – evaluated from left to right. Other orders of operations can be enforced by using parentheses, **`(`** and **`)`**:

```
>> (10 - (13 + 4 * 2)) / 5
ans =
   -2.2000
```

Variables: Variable names in MATLAB are combinations of letters and digits, beginning with a letter. For example, **`x`**, **`yFinal`**, and **`b25`** are all valid variable names; **`1zfirst`**, **`x%3`**, and **`y-total`** are invalid names. Names are also case-sensitive, so that **`xyz`** is different from **`xYz`**. Consider the example:

```
>> x = 4.5
x =
   4.5
```

Now, the value 4.5 is stored in the variable named x which can be used in subsequent calculations:

```
>> y = x^3 - 2/x + 1
y =
   91.6806
```

MATLAB commands and variable names are case-sensitive! Notice the message generated when the undefined variable Y (an uppercase y!) is referenced

```
>> Y^2 - 3

??? Undefined function or variable 'Y'.
```

Combining commands and suppressing output: The comma (`,`) and semi-colon (`;`) are characters that have very special meaning in MATLAB, and will prove to be very useful:

- The comma operator is used to group multiple commands on the same line, for example:

```
>> x=3.5, y=-5.0, x^3 - y
x =
   3.5000
y =
   -5
ans =
   47.8750
```

- The semi-colon operator can be used similarly to the comma, but it also acts to suppress display of output. If the previous example were slightly modified to the following

```
>> x=3.5; y=-5.0; x^3 - y
ans =
   47.8750
```

  then the results of the first two assignment statements are suppressed and only the last is shown.

Pre-defined constants: MATLAB defines several useful constants including:

- `pi`, $\pi = 3.14159265\ldots$
- `i` and `j`, both equal to the imaginary unit, `sqrt(-1)`
- `inf`, "infinity" or ∞
- `NaN`, "not-a-number"
- `ans`, which is always assigned to be the the result of the previous command

You should avoid re-assigning values to the above constants, if at all possible. The possible exceptions are `i` and `j`, both of which are commonly used as loop indices: reassigning these values is acceptable since the complex unit can always be obtained using `sqrt(-1)`. Note that the constant $e = 2.7182818\ldots$ (the base corresponding to the natural logarithm) is not predefined in MATLAB but can always be obtained using `exp(1)`.

Built-in functions: Just as in many other high-level languages, MATLAB implements more complex arithmetic functions as built-in procedures. These include the square root (`sqrt`), exponential (`exp`), various logarithms (`log`, `log10`, `log2`), absolute values (`abs`), and trigonometric functions (`sin`, `cos`, `tan`, `atan`, . . .). For example,

```
>> sin(45)
ans =
   0.8509
```

returns the sine of 45 radians . . . what you probably intended was to compute the sine of $45^\circ = \frac{45\pi}{180}$ radians:

```
>> sin(45/180*pi)
ans =
   0.7071
```

and then check that you get $\frac{\sqrt{2}}{2}$ as expected:

```
>> sqrt(2)/2
ans =
   0.7071
```

Notice that all MATLAB calculations introduce rounding error, which sometimes show up in unexpected ways. For example, you shouldn't be confused when you see

```
>> tan(pi)
ans =
   -1.2246e-16
```

Keeping in mind that rounding errors are ubiquitous, and that -1.2246×10^{-16} is very very close to zero, just interpret this result as $\tan \pi = 0$.

Type `help elfun` for a complete list of built-in functions.

Display formats: The default format for display of double precision numbers is using 5 significant figures, as in the previous examples. If more precision is desired, you can use MATLAB's `format` command to alter the way output is displayed; in particular, `format long` increases the accuracy with which the results are displayed while and `format short` returns to the default. The following two statements

```
>> format short, exp( log(10000) )
ans =
   1.0000e+04
```

return the expected value of 1, but using the "long" format

```
>> format long, exp( log(10000) )
ans =
   1.000000000000001e+04
```

many more digits are displayed . . . and the round-off errors inherent in the floating point calculations for `log` and `exp` become evident! Type `help format` to see a complete list of possible output formats.

MATLAB data types: There are several basic data types in MATLAB, including:

- *Integers:* such as `-5` or `93456`.
- *Double precision reals:* in MATLAB, all real numbers are stored in double precision, unlike languages such as C or Fortran that have a separate `float` or `real*8` type for single precision reals. An **extremely useful short form** for entering very large or very small real numbers is the "e" notation, in which `-1.23456e-7` is short for -1.23456×10^{-7}, and `9.23e+12` stands for 9.23×10^{12}. Some examples:

  ```
  >> -1.23e-2
  ans =
     -0.0123
  >> 7e9
  ans =
     7.0000e+09
  ```

- *Complex numbers:* which are entered as `3+2*i` or equivalently `3+2*sqrt(-1)`.
- *Strings:* which are arrays of characters entered as `'a'` or `'This is a string'`.

This covers only the data types that you are most likely to require in this course. Typing `help datatypes` gives a more complete list.

4 Vectors

The fundamental data structure in MATLAB is a *double precision matrix*. While it is not obvious from the previous examples, even scalars have an internal representation as 1×1 matrices or arrays.

In this section, we will introduce examples of *vectors*, which are matrices of size $1 \times n$ or $n \times 1$, and talk about the more general case of $m \times n$ matrices in Section 5.

The basic syntax for entering a vector is as a list of values surrounded by square brackets, [ ]. For example:

```
>> v1 = [ 4.5 3 -1.2 ]
v1 =
   4.5000    3.0000   -1.2000
>> length(v1)
ans =
   3
```

The variable v1 is now a row vector of length 3 or a matrix of dimension 1×3. In the above example, spaces were used to separate entries in the vector; an equivalent way to define the same vector is using a *comma-separated* list of values: **v1 = [4.5, 3, -1.2]**.

Column vectors, which have dimension $n \times 1$, can also be entered using a similar syntax, the only difference being that entries are separated by semi-colons:

```
>> v2 = [ 4; -5; 2 ]
v2 =
   4
  -5
   2
>> length(v2)
ans =
   3
>> size(v1), size(v2)
ans =
   1    3
ans =
   3    1
```

Notice how the **length** command returns the length of the row or column vector, while the **size** command returns the dimensions of the vectors, interpreted as matrices.

Subscripting vectors: Individual entries within a vector can be accessed or changed using the **subscripting** operation.[2] In MATLAB, the ith entry of a vector v is represented using the notation **v(i)**, where the subscript is in parentheses. Note that all vectors are indexed starting from entry **i=1**, unlike C for example. The second entry of vector v2 can be displayed using

```
>> v2(2)
ans =
    -5
```

and then changed via the command

```
>> v2(2) = 5
v2 =
     4
     5
     2
```

The colon operator: A useful syntax for generating vectors containing regularly-spaced values is the colon notation, **a:b:c**, which produces a list of real numbers (actually, a row vector) starting from a, ending at c, and incremented by b. If the middle entry b is omitted, then a spacing of 1 is assumed between the numbers. Consider

[2]This section describes only the *very simplest* subscripting operations. To see the true flexibility and power of MATLAB's vector/matrix subscripting, take a look at http://www.mathworks.com/company/newsletters/articles/matrix-indexing-in-matlab.html, an article from the September 2001 issue of *Matlab Digest*.

the following examples:

```
>> 1 : 4
ans =
   1     2     3     4
>> 1 : 0.1 : 1.5
ans =
  1.0000    1.1000    1.2000    1.3000    1.4000    1.5000
>> 10 : -3 : 1
ans =
   10     7     4     1
>> z = 1 : 0
z =
   empty matrix: 1-by-0
```

Notice the last example, which assigns the **empty matrix** to variable `z`; the same result can be obtained with the command `z = []` (this is a nice way to "unassign" a variable).

Vector operations: Vectors can be added or subtracted from each other, provided that they have compatible dimensions. With vectors `v1` and `v2` defined above, the following command generates an error message:

```
>> v1 + v2
??? Error using ==> +
Matrix dimensions must agree.
```

because one is a row vector and the other is a column vector. However, if we define:

```
>> v3 = [ 4, -5, 2 ];  v1 + v3
ans =
   8.5000   -2.0000    0.8000
```

then the vectors are of compatible sizes and so their sum represents a valid operation.

In order to perform the same sort of **element-wise** calculations using multiplication, division and exponentiation, MATLAB has introduced the operators `.*`, `./` and `.^`:

```
>> v1 .* v3
ans =
   18.0000   -15.0000   -2.4000
>> v1 ./ v3
ans =
   1.1250   -0.6000   -0.6000
```

The exponentiation operator can be employed in two ways, with either a scalar or vector exponent:

```
>> v1 .^ 2,  v1 .^ v3
ans =
   20.2500    9.0000    1.4400
ans =
  410.0625    0.0041    1.4400
```

The reason for MATLAB needing to define these two "dot"-operators will become clearer later on in Section 5: in particular, `*` should actually be thought of as performing matrix multiplication that reduces to the dot product when the two arguments are vectors. These are a few examples of *operator overloading* that is possible in object-oriented languages like C and Java.

All of MATLAB's built-in arithmetic functions are also designed to operate on vectors (and matrices) so that we can construct algebraic expressions that operate on vectors element-wise; for example, the following code computes

$2\sqrt{x} + \frac{x}{y} - x^3 \cos(\pi y)$, element-wise, for each element in vectors x and y:

```
>> x = [ 1 2 3]; y = [-1 4 2]
>> 2 * sqrt(x) + x ./ y - x .^ 3 .* cos(pi*y)
ans =
     2.0000    -4.6716    -22.0359
```

Notice how scalar operations are interspersed with element-wise ones in this expression; for example, `2 * sqrt(x)` is well-defined as the product of a scalar with a vector; whereas `x/y` is not, and `x./y` must be used instead.

5 Matrices and linear algebra

Matrices are the basic data structure in MATLAB, and as mentioned earlier, vectors are just a special type of matrix having dimension $1 \times n$ or $n \times 1$. Typing `help elmat` and `help matfun` will present a list of the many matrix commands and functions available in MATLABof which only a few will be discussed here.

Defining and initializing matrices: The syntax for defining matrices is very similar to what you've already seen for vectors. Spaces (or commas) separate elements within a row, and semi-colons indicate the start of a new row. Consequently, typing

```
>> A = [ 2 -1 0 0; 1 1 2 3; -1 4 0 5 ]
```

initializes a 3×4 matrix and yields the MATLAB output

```
A =
      2    -1     0     0
      1     1     2     3
     -1     4     0     5
```

so that the variable A now contains a 3×4 matrix.

Individual entries within a matrix can be accessed or changed in the same way as for vectors; for example, the command `A(3,2) = 0` replaces the entry 4 in the last row with a zero.

Special matrices: There are several commands that initialize matrices of special type, for example:

- `zeros(n,m)`, for an $n \times m$ zero matrix,
- `ones(n,m)`, for an $n \times m$ matrix containing all ones,
- `eye(n)`, for the $n \times n$ identity matrix.

To initialize a square special matrix, you can use the short form `zeros(n)` which assumes there are the same number of columns as rows.

Basic matrix operations: The operators `+`, `-`, and `*` implement matrix addition, subtraction and multiplication. For example, if we first define the matrices

```
>> A = [ 0 2; 1 3; 4 -1 ];
>> B = [ 1 1; 0 0; -1 2 ];
>> C = [ 1 0 1; -2 3 -1 ];
```

then the following commands yield

```
>> A + B
ans =
      1     3
      1     3
      3     1
```

and

```
>> A * C
ans =
    -4     6    -2
    -5     9    -2
     6    -3     5
```

while the following multiplication generates an error because the inner dimensions of matrices A and B don't match

```
>> A * B
??? Error using ==> *
Inner matrix dimensions must agree.
```

Another essential matrix operation is the **transpose**, which in MATLAB is performed using the single quote character,

```
>> D = A'
D =
     0     1     4
     2     3    -1
```

or alternately `D = transpose(A)`. Notice that now the multiplication `D*B` is well-defined, since D is 2×3 and B is 3×2:

```
>> D * B
ans =
    -4     8
     3     0
```

Of course, the same result could have been obtained by simply typing `A' * B`.

Element-wise matrix operations: Just as with vectors, the element-wise operations `.*`, `./` and `.^` can also be applied to matrices. I hope the following examples (using the matrices defined on page 8) are self-explanatory:

```
>> A .* B
ans =
     0     2
     0     0
    -4    -2
>> B ./ A
Warning: Divide by zero.
ans =
       inf    0.5000
         0         0
   -0.2500   -2.0000
>> B .^ 2
ans =
     1     1
     0     0
     1     4
```

Notice in the second example that a division-by-zero error in the $(1, 1)$ entry generates a warning message and yields a value of `inf` for the operation $1/0$.

Solving linear systems: The command for solving a linear system in MATLAB is simple, but requires a little getting used to. Suppose you have defined a matrix A and right hand side vector b as follows:

```
>> A = [2 -1 3; -4 6 -5; 6 13 16];
>> b = [13; -28; 37];
```

The solution to the linear system $Ax = b$ (that is, $x = A^{-1}b$), is found using the **matrix left divide operator**, \:

```
>> x = A \ b
x =
    3.0000
   -1.0000
    2.0000
```

Another useful command is **`[L, U] = lu(A)`**, which performs the LU factorization of the matrix `A` and stores the corresponding lower and upper triangular matrices in `L` and `U` (see **`help lu`** for more information).

Other useful matrix commands: For square matrices, `A`, the following commands are extremely helpful:

- **`inv(A)`**, the matrix inverse,
- **`det(A)`**, the determinant of a matrix,
- **`trace(A)`**, the trace,
- **`cond(A)`**, the condition number,
- **`norm(A)`**, the matrix norm.
- **`eig(A)`**, finds the eigenvalues and eigenvectors.

6 Other built-in functions:

There are a **huge** number of functions that are built-in to MATLAB (we have already seen examples such as **`sin`**, **`abs`**, **`zeros`**, **`trace`**, etc.). Because there are so many, I won't make any attempt to describe them all here. You can get a list of the functions of various sorts that are available to you by typing **`help xxx`** where **`xxx`** is one of the following:

`elfun`	elementary math functions.
`specfun`	specialized math functions.
`matfun`	matrix functions - numerical linear algebra.
`datafun`	data analysis and fourier transforms.
`polyfun`	interpolation and polynomials.
`funfun`	function functions and ode solvers.
`sparfun`	sparse matrices.
`strfun`	character strings.
`iofun`	file input/output.
`timefun`	time and dates.

7 Loops

MATLAB does have the useful **`for`** and **`while`** loop constructs (use **`help`** to see the syntax). However, one thing that should stand out about MATLAB code is the relative scarcity of loops. MATLAB's operations and built-in functions naturally act on vector and matrix arguments and so minimize the need for these loop constructs. For example, compare the following two pieces of code (the one on the left written in MATLAB, and the right in C) that both initialize an array of points, `x`, and then compute $3 + x^2e^x$:

```
x = [ 0 : 0.1 : 10 ];
y = 3 + x .^ 2 .* exp(x)
```

```
int i;
double x[101], y[101];

for( i = 0; i <= 100; i++ ) {
  x[i] = i * 0.1;
}

for( i = 0; i <= 100; i++ ) {
  y[i] = 3 + x[i] * x[i] * exp(x[i]);
  printf( "%f ", y[i] );
}
```

Notice the following:

- MATLAB requires no declarations for variables.
- the colon operator, "`:`", is a very simple way to replace initialization loops.
- the element-wise or "dot" operators (`./` and `.^`) eliminate the need for the second `for` loop.
- leaving out the semi-colon at the end of second MATLAB assignment statement has the same effect as the `printf` command in the C code ... not only is MATLAB code more compact, but by *shortening* your code you can do *even more*! ⌣̈
- MATLAB is *FAST!* This is not obvious from just looking at the code on the left, but MATLAB is *optimized* for operations on vectors, and will naturally execute faster that the "equivalent" C code on the right (depending on your computer's architecture, of course).

8 Command history and editing

A helpful feature of MATLAB's command line is the ability to recall previous commands. MATLAB keeps a record of your command "history," and allows you to recall these old commands through the use of the up and down arrow keys on your keyboard (↑ and ↓). This is *extremely useful* in situations where you repeat a command several times in succession, or if you make a mistake and need to make a minor change to what you've just typed.

Hitting the up arrow key will recall the last command you entered, and pressing it more than once will recall older commands in your session "history." You can modify previous commands by using the left and right arrow keys (← and →) to move to the appropriate spot in the previous command and edit what you have written. Then simply hit **`Enter`** again to execute the repeated or modified command.

9 Saving your commands in "script m-files"

Once you move beyond the simplest of MATLAB calculations, you will find that you are entering much longer sequences of commands. When you need to re-enter such a sequence, the "arrow-key" method of recalling commands is no longer convenient.

Just like most other programming languages, MATLAB allows you to store sequences of commands in a separate file, which you can think of as a MATLAB *program* that can be executed much more conveniently by referring to the file instead of typing the individual commands. A MATLAB program is stored in a file with a "`.m`" extension and is referred an "m-file."

For example, suppose you created a file called "`mycmd.m`" containing the following three lines (from the example at the end of Section 4):

```
x = [ 1 2 3];
y = [-1 4 2];
2 * sqrt(x) + x ./ y - x .^ 3 .* cos(pi*y)
```

Then, instead of typing each individual command as you had to do before, you can simply type the name of your m-file at the MATLAB prompt, omitting the "`.m`" extension:

```
>> mycmd
ans =
   2.0000   -4.6716   -22.0359
```

Voilà!

Note on file locations: MATLAB needs to know where your m-file is located; in other words, any m-file you want to execute must be located in your *current working directory* or *path*. Therefore, it is easiest if you save all of your m-files in a single directory, say `f:\macm316\matlab\`, so that if you begin every MATLAB session with the "`cd`" command to change to this directory (see the discussion of files and directories on page 2), then your m-files are always available.

Comments: You will find when you write longer and more complicated pieces of MATLAB code that comments will become very useful. Comments are delimited by the `%` (per cent) character: anything occurring on a line after a `%` is ignored. Consider the following:

```
>> % define two vectors
>> v1 = [ 4.5, 3, -1.2 ]; v3 = [ 4, -5, 2 ];
>> v1 .* v3      % calculate their dot product
ans =
   18.0000   -15.0000   -2.4000
```

Helpful hint: *It is always a good idea to generously comment your assignment submissions so that your marker understands clearly what you are doing!!!*

10 Inline functions

The script m-files described in the previous section are not "functions" by definition; they have no input or output arguments, and they simply execute a sequence of MATLAB commands on the variables defined in the workspace. Nonetheless, MATLAB does support functions which can come in one of two flavours: inline functions and m-files. The first of these will be described next.

The simplest way of defining a function is as an *"inline function"*. The following command defines a function $f(x) = x\sin(x) - 3$ and then evaluates $f(3)$ by passing the function to MATLAB's `inline` command within single quotes:

```
>> f = inline( 'x*sin(x)-3' ), f(3)
f =
     inline function:
     f(x) = x*sin(x)-3
ans =
   -2.5766
```

Functions of several variables can be defined in a similar manner, where additional arguments to `inline` are used to indicate the ordering of the independent variables in the function's argument list:

```
>> g = inline( 'x^3 + y^3/3 - y', 'x', 'y' ), g(3,2)
g =
     Inline function:
     g(x,y) = x^3 + y^3/3 - y
ans =
   27.6667
```

Warning: *Keep in mind that only the simplest of functions can be defined using* `inline`.

11 Function m-files

For more complicated functions, such as those involving loops, conditionals, etc., you must define your function using an m-file. "Function m-files" differ from "script m-files" described in the previous section in that they contain a **function** definition line, through which are passed input and output arguments. For example, consider the following sequence of commands which are saved in a file named "myfunc.m":

```
function [area] = myfunc( a, b, c )
% Compute the area of a triangle with side lengths a, b and c.

% First, check the input parameters, and exit if invalid:
if( a < 0     | b < 0     | c < 0     |   ...
    (a+b) < c | (b+c) < a | (c+a) < b )
  disp('** Error **   Side lengths do not define a triangle.')
  return
end

% If everything is OK, then compute the area:
s = (a + b + c) / 2;
area = sqrt( s * (s-a) * (s-b) * (s-c) );
```

This function computes the area of a triangle with side lengths a, b and c using the well-known "Heron's formula"[3]. Provided the file "myfunc.m" is in the *current working directory* (see Page 2), the function can be invoked as follows:

```
>> a = myfunc(2, 3, 4)
a =
    2.9047
```

Invoking **myfunc** with invalid side lengths demonstrates how the error-checking works:

```
>> a = myfunc(2, 3, 6)

*** Error in myfunc ***   Side lengths do not define a triangle.
```

Notice the following:

- the name specified in the **function** statement on the first line of the file must match the m-file name ("myfunc.m").
- the **return** statement is used to exit the function.
- try typing **help myfunc** at the MATLAB prompt: the second line of the m-file (the comment line) is displayed ... this feature is **very useful** for documenting your m-files!

The following two examples demonstrate how the syntax for the calling sequence of MATLAB functions differs from other languages such as C and FORTRAN:

In the C language:	*In the FORTRAN language:*
`double myfunc( a, b, c ) {`	`real*8 function myfunc( a, b, c )`
`double a, b, c;`	`real*8 a, b, c`
`...`	`...`
`return( sqrt(s*(s-a)*(s-b)*(s-c)) ); }`	`return( sqrt(s*(s-a)*(s-b)*(s-c)) )`

Passing multiple output arguments: While having multiple **input** arguments to a function is common, multiple **output** arguments are not as common, but MATLAB handles these quite simply. For example, suppose you wanted the option of returning the intermediate variable s along with the area. A simple way to do this is to replace the **function** statement with the following:

```
function [area, s] = myfunc( a, b, c )
```

[3]Reference: http://www.mste.uiuc.edu/dildine/heron/triarea.html

at which point you can call the function by typing the following command at the MATLAB prompt:

```
>> [a,s] = myfunc( 2, 3, 4 )
a =
    2.9047
s =
    4.5000
```

Note: *If you omit the second output argument in the function call, using* **`a = myfunc(2,3,4)`** *as before, then* MATLAB *is smart enough to ignore the* `s` *value and only assign the area to the variable* `a`.

Passing functions as parameters: Suppose you need to write a simple function m-file that takes a function $f(x)$ and an interval $[a, b]$ and computes the average value $\frac{1}{2}(f(a) + f(b))$. In this case, your function m-file must take three arguments: the two endpoints, a and b **and the function f**! MATLAB makes passing functions as arguments very easy using the **`feval`** function, which evaluates a function at a given point. Consider the following two-line m-file "`favg.m`":

```
function avg = favg( f, a, b )
avg = 0.5 * (feval(f,a) + feval(f,b));
```

This function can be invoked in one of several ways:

1. passing a built-in function as a string,

   ```
   >> favg( 'sin', 0, pi/4 )
   ans =
       0.3536
   ```

2. passing an inline function (which behave like built-in functions in this respect),

   ```
   >> f = inline('x^3+2*x-1/2');
   >> favg( f, 1, 5 )
   ans =
      68.5000
   ```

3. passing the name of a function m-file as a string,

   ```
   >> favg( 'myf', 1, 5 )
   ans =
      68.5000
   ```

   where in this example "`myf.m`" contains the lines:

   ```
   function fval = myf( x )
   fval = x^3+2*x-1/2;
   ```

4. one other way of passing functions as parameters is using *string-functions*, as is done in Section 12 with **`ezplot`**.

12 Plotting

There are two ways to generate plots in MATLAB: as functions or as lists of points, and each uses a different plotting routine.

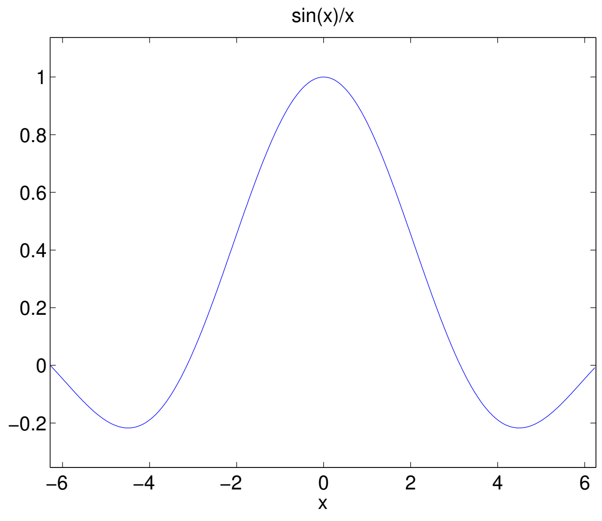
Plotting functions, with `ezplot`: The **`ezplot`** command is a bare-bones plotting routine that takes a function as an argument, and the simplest way to define a function in MATLAB is as a string. For example, to plot the function $f(x) = \sin(x)/x$, we would use the command:

```
>> ezplot( 'sin(x)/x' )
```

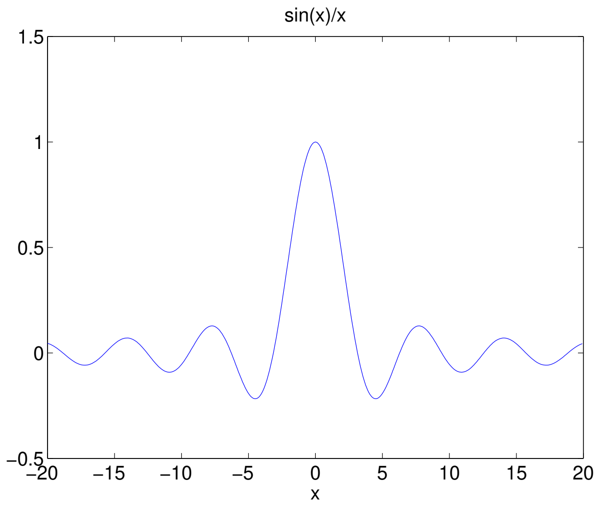
which pops up a separate figure window containing the plot in Figure 1(a). To expand our view of the plot, we can provide an optional second argument to `ezplot`, which is a vector of axis limits that can be specified either as `[xmin, xmax]` or `[xmin, xmax, ymin, ymax]`. For example,

```
>> ezplot( 'sin(x)/x', [-20 20 -0.5 1.5] )
```

produces the plot in Figure 1(b). Type `help ezplot` for complete information, and more examples.



(a) `ezplot` with no axis limits.



(b) `ezplot` with axis limits `[-20 20 -0.5 1.5]`.

Figure 1: Plots of $f(x) = \sin(x)/x$ using `ezplot`.

Plotting lists of points, with `plot`: MATLAB's do-everything, all-purpose plotting command is `plot`, and we are only going to touch the tip of the iceberg here in terms of what it can do.

`plot` is designed to generate graphics of lists of points. For example, we can define a list of x–values and compute a corresponding list of y–values for the function $f(x) = \arctan(\sin x) + \sin(\arctan x)$:

```
>> x = [ -20 : 0.01 : 20 ];
>> y = atan(sin(x)) + sin(atan(x));
```

> ***Note:*** *our judicious use of the semi-colon operator to suppress the output from both commands … each vector has 4001 elements, which we'd definitely prefer not to see!!!*

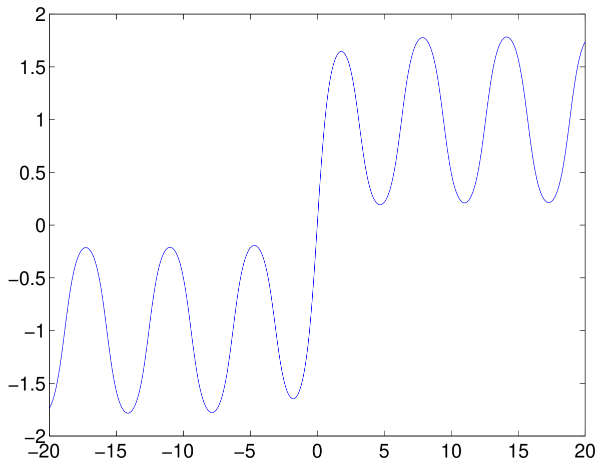
The list of points can then be plotted using the command `plot( x, y )`, with the results displayed in Figure 2(a). MATLAB provides many plotting features that allow the creation of more complicated plots. For example, using the following commands, we can add a grid and labels to the plot:

```
>> grid on
>> xlabel('x'), ylabel('y')
>> title('My first plot')
```

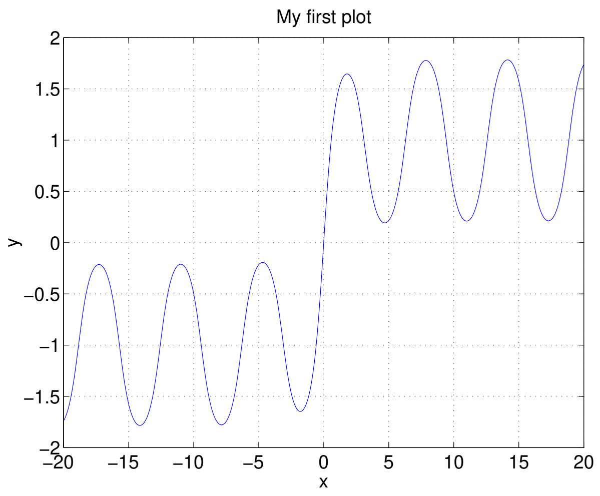
with the updated figure being displayed in Figure 2(b) Finally, we can add the plot of a second function on the same axes, using `hold on` to prevent MATLAB from clearing the axes before displaying the results of the second `plot` command:

```
>> hold on
>> plot( x, atan(x), 'r--' )
>> legend( 'f(x)', 'arctan(x)', 0 )
>> hold off
>> axis( [-25 25 -3 3] )
```

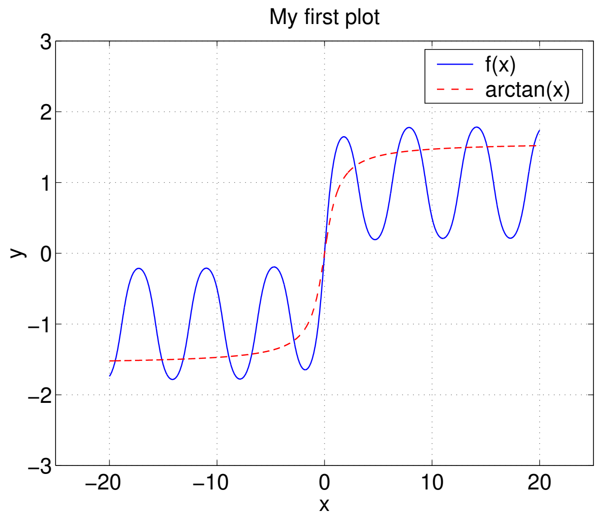
The resulting figure is displayed in Figure 2(c). Notice the use of the following new plotting features:



(a) Plot of $f(x) = \arctan(\sin x) + \sin(\arctan x)$.



(b) Same plot with title, axis labels, and a grid.



(c) Plotting two functions on the same axes.

Figure 2: Examples of the `plot` command.

- the **`legend`** command, which adds a "legend" or "key" to the plot which is very helpful in distinguishing one curve from another.
- giving a color and line type **`'r--'`** as the third argument to the **`plot`** command, where the **`r`** indicates a red line and **`--`** means a dashed line. The default line type is a solid blue line, **`'b-'`**.
- the **`axis`** command, which explicitly sets the x– and y–axis limits for the plot, using the format ***`[xmin, xmax, ymin, ymax]`*** for the argument. This is particularly useful when MATLAB's default choice of plotting axes isn't suitable or pleasing to the eye.

For more information about MATLAB's plotting capabilities, line types and other useful commands, just type **`help plot`**.

Bibliography

[1] Adrian Biran and Moshe Breiner, "MATLAB 5 for Engineers," 2nd edition, Addison-Wesley, 1999.

[2] Graeme Chandler, "Introduction to MATLAB," The University of Queensland, 2000 (URL: `http://www.maths.uq.edu.au/department/computing/matlab/mlb_intro.html`).

[3] David F. Griffiths, "An Introduction to MATLAB," Version 3.1, University of Dundee, 2015 (URL: `http://www.maths.dundee.ac.uk/software/MatlabNotes.pdf`).

[4] Andrew Knight, "Basics of MATLAB and beyond," Chapman & Hall, 2000.

[5] The MATLAB home page, The Mathworks (URL: `http://www.mathworks.com`).

[6] Gerald Recktenwald, "Numerical Methods with MATLAB: Implementation and Application," Prentice-Hall, 2000.

[7] Kermit Sigmon, "MATLAB Primer," 3rd edition, University of Florida, 1993 (URL: `http://web.mit.edu/6.777/www/downloads/primer.pdf`).