CHARLES UNIVERSITY
Faculty of mathematics
and physics

Introduction to MATLAB

Winter Semester 2019

Scott Congreve

Charles University,
Faculty of Mathematics and Physics,
Department of Numerical Mathematics,
Sokolovská 83, 18675 Praha

Contents

List of Code Examples

List of Figures

1 Introduction

MATLAB is a high-level programming language and interactive environment designed for numerical computation. It has both an interactive console for executing individual commands, and support for writing full scripts (programs).

In this section, we shall give a brief overview of the MATLAB program. In Sections 2–5 we shall cover various basic MATLAB commands, using only the interactive console. Section 6 will cover using MATLAB to write scripts and functions. Finally, Sections 7 & 8 will cover a couple of more advanced topics.

1.1 Overview of the UI

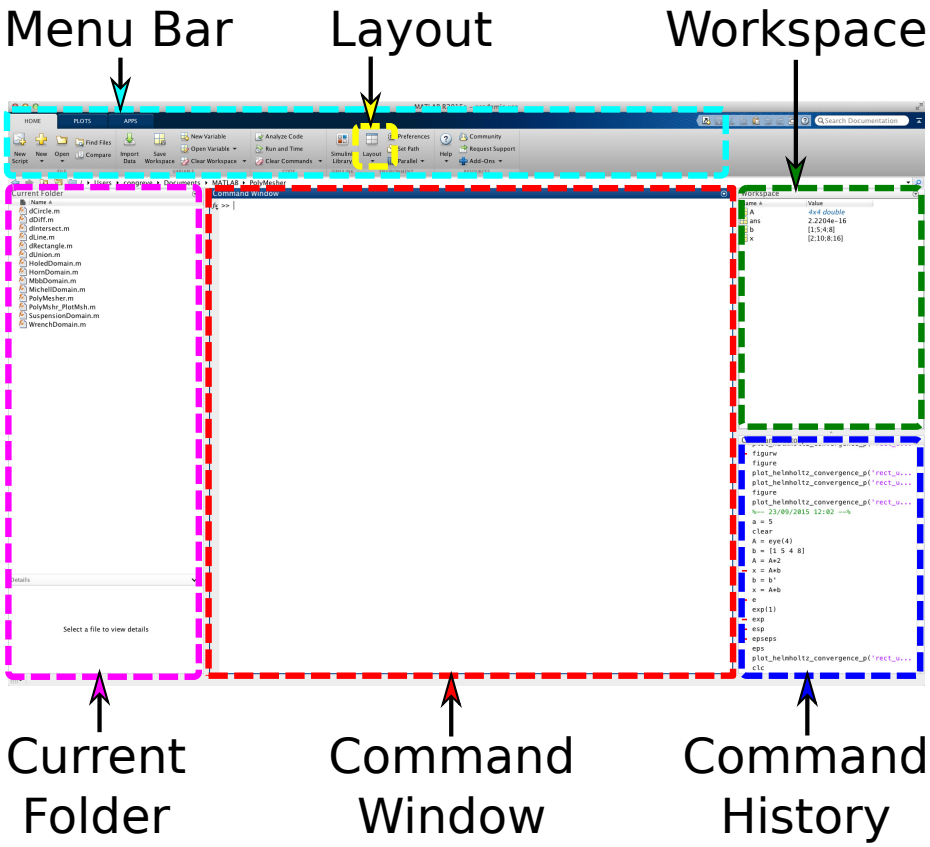

Figure 1: Overview of the User Interface

Upon launching MATLAB the main window is displayed. The main area of this window is usually the *Command Window*, this is an interactive console where MATLAB commands can be entered at the prompt and results seen immediately. The default MATLAB window also displays a *Workspace* panel, which lists all the current variables (see Section 2.3), and a *Current Folder* panel listing all files in the directory MATLAB considers "current" (see Section 1.2). Another useful panel, which is often not shown by default, is the *Command History* panel, which lists all recent commands entered into the *Command Window*. This can be shown from the Layout button on the Home tab of the menu bar.

The MATLAB window is completely customisable. Panels can be shown or hidden from the Layout button on the Home tab of the menu bar. Panels can also be dragged to different positions, placed into tabs with each other and even *undocked* into separate windows. Each panel has a

small downward pointing arrow in the top right corner, which opens a menu containing various customisation options.

1.2 Basics of the Command Window

The *Command Window* consists of a prompt (>>) at which MATLAB commands can be entered. Results from each MATLAB command ran is also displayed in the *Command Window*. You can clear the current command window of all output by entering the `clc` command into the *Command Window*.

MATLAB keeps a history of all commands entered (these can be seen and selected to run again from the *Command History* panel). You can also use the ↑ and ↓ arrow keys on the keyboard to scroll backwards or forwards, respectively, through the history of commands entered. If you start to type a command and then press the ↑ or ↓ arrow keys then MATLAB will only scroll through commands which start with the text already entered.

When you try to execute a MATLAB function it searches in a list of paths for the a file containing the definition of that function. By default this list consists of a set of built-in MATLAB directories and also the *current directory* according to MATLAB. Usually when you start MATLAB this will be the *MATLAB* subdirectory in your HOME or Documents folder. When we come to write scripts and functions (see Section 6) your current directory will need to be the same as the directory where you save these files in order to be able to run them. Entering the command, `cd`, on its own lists the current directory. You can also change the current directory by using

```
>> cd path
```

where ***path*** is the path to change to. Directories in a path are separated by a `/` and a special `..` directory can be used to change to the *parent directory*. For example, if my current directory is `Users‣congreve‣Documents‣MATLAB` then calling

```
>> cd ../TestFolder
```

will change the current directory to `Users‣congreve‣Documents‣TestFolder`. Note that if any folder contains a space you should surround the path with single quotation marks:

```
>> cd '../Folder With Space/Folder'
```

You can see a list of all files in the current directory with the `ls` command, or only MATLAB specific files with the `what` command.

To exit MATLAB type `exit` at the prompt.

1.3 Documentation & Help

MATLAB has built-in documentation for all its commands, functions and syntax. This documentation can be viewed in two different ways. The first way is a graphical help window which can be launched by selecting the Help ⟩ Documentation menu, or by entering the command `doc` into the *Command Window*. The `doc` command can be followed by a name (usually of a MATLAB function), in which case the documentation window is automatically launched to view the documentation for that command. For example,

```
>> doc sin
```

will open the documentation for the `sin` function.

The second method is a text-based help displayed directly in the *Command Window*. To view the contents for this help type `help` into the *Command Window*. Again you may add a name of a function, command, toolbox, etc. to display the help for that command.

```
>> help sin
 sin    Sine of argument in radians.
    sin(X) is the sine of the elements of X.

    See also asin, sind.

    Other functions named sin
    Reference page in Help browser
       doc sin
```

Code Example 1: `help` Example

2 Basic Mathematics

MATLAB is designed to perform mathematical operations on scalars, vector and matrices. We shall start by looking at the basic scalar mathematics.

2.1 Scalar Arithmetic

In MATLAB you can enter mathematical statements to solve in an ALMOST identical way to how you write them on paper. MATLAB supports five basic scalar mathematical operators. These are `+` (for addition), `-` (for subtraction), `*` (for multiplication), `/` (for division), and `^` (for raising to a power). There is also a left division operator `\` which divides the second term by the first. Using these basic commands you can use MATLAB as a calculator; i.e., entering

```
>> 5^2+9.5-11*2
ans =
   12.5000
```

displays the result of $5^2 + 9.5 - 11 \times 2$. MATLAB follows the basic mathematical rules for precedence; `^` is evaluated first, then `*` and `/`, and then `+` and `-`. Operators of the same precedence are evaluated with left-to-right associativity — the first operator from the left is evaluated first. Brackets `( )` can be used to specify order of evaluation.

```
>> 2^2^3
ans =
    64

>> 2^(2^3)
ans =
   256
```

Code Example 2: Evaluating $(2^2)^3$ and 2^{2^3}

Note that unlike in normal mathematics the multiplication symbol must be used wherever multiplication is required.

```
>> 2(4+5)
 2(4+5)
  |
Error: Unbalanced or unexpected parenthesis or bracket.

>> 2*(4 + 5)
ans =
    18
```

Code Example 3: Demonstration of requirement for multiplication symbol

2.2 Number Format & Special Constants

By default in MATLAB all numbers generated are of *double* type. The technically of what this means exactly is beyond the scope of this course, but essentially each number is stored with the computer's memory as a 64-bit *binary* floating-point number. This means that it can represent floating-point numbers, but we do have to allow for small rounding errors in computations as not all decimal floating point numbers can be accurately represented within the number of bits (the only real important point about this we shall discuss in Section 6.3).

MATLAB, like most programming languages, allows us to enter numbers in an exponential (base 10) form. Essentially entering `1.5e-10` is short-hand for 1.5×10^{-10}, and `7.95e5` is short-hand for 7.95×10^5.

When MATLAB displays a floating point number it usually displays it in *short* form (four decimal places) in either normal or exponential form.

```
>> 190.2
ans =
  190.2000

>> 1909.205
ans =
   1.9092e+03
```

Code Example 4: Short format for real numbers

Notice that in the last case we lost some of the number in the display (it is still there but MATLAB has not displayed the result). We can ask MATLAB to display all results in *long* form (15 decimal places) with the command `format long`, and we can switch back to short format with `format short`. More formats exist as well, use `help format` to see the complete list. You can set the default `format` in the *Preferences* window in MATLAB (under MATLAB 〉 Command Window).

```
>> format long
>> 19.2
ans =
  19.199999999999999

>> 1909.205
ans =
     1.909205000000000e+03
```

Code Example 5: Long format example; also demonstrates rounding error

MATLAB has a number of built-in constants and special values that can be used (and displayed). `pi` returns the constant value for π while `eps` returns the difference between 1 and next largest double-precision floating-point number. Double-precision numbers have three special numbers, `inf`/`Inf` and `-inf`/`-Inf` represent ∞ and $-\infty$, respectively, while `nan`/`NaN` represents a special *Not a Number* value.

```
>> 0/0
ans =
   NaN
>> 1/0
ans =
   Inf
>> -1/0
ans =
  -Inf
>> pi
ans =
   3.141592653589793
>> eps
ans =
     2.220446049250313e-16
```

Code Example 6: MATLAB constants/special values

2.3 Variables

As a programming language MATLAB has a concept of variables that be used to store values. Assigning a value to a variable is done via the assignment = operator. When assigning a variable, the value stored into the variable is output in the *Command Window*; this can be suppressed by ending the command with a semicolon (;). Variables can be used in expressions similar to basic mathematics and entering a variable name on its own at the prompt (without a trailing semicolon at the end) will output the variables value.

```
>> x = 2^2
x =
     4

>> 5*x+9
ans =
    29

>> y = 9;
>> y
y =

     9
```

Code Example 7: Definition and usage of the variables x and y

Note again that we need to explicitly use the multiplication operator for calculating $5x + 9$.

Variables names in MATLAB must start with a letter and can contain only letters, numbers and the underscore (_) character. Note that by letter we mean the basic 26 *English* letters (so no accented letters). Ideally, variable names should be self-explanatory where possible and also note that all variable names are *case sensitive*; i.e., `A` and `a` are different variables. Note that defining a variable with the same name as a built-in MATLAB constant (`pi`, `eps`, etc.) or functions will hide the definition of that function or constant, so this should be avoided. There is also a special variable called `ans`, which stores the result of the last command entered if the result is not saved to a variable.

When working in the *Command Window* all variables defined are saved in the *Workspace*. You can see the list of all variables in the *Workspace* panel or by entering the `who` or `whos` command. You can clear all variables from the current workspace with the `clear` command; alternatively, you can clear a single or list of variables by enter the names of the variables after the `clear` command.

```
>> clear x y
```

Code Example 8: Clearing only the x and y variables from the workspace

2.4 Complex Numbers

MATLAB supports complex numbers as well as real numbers. To specify an imaginary number you use `i` or `j` either directly or as a suffix to a number. For example, to generate the complex number $z = 5 + 4i$:

```
>> z = 5+4i
z =
   5.0000 + 4.0000i
```

Code Example 9: Definition of complex number

When used in scripts (see Section 6) newer versions of MATLAB will produce warnings about using `i` or `j` without a number prefix and will advise the use of `1i` and `1j` instead.

2.5 Functions

MATLAB has a large collection of built-in functions for mathematical operations. Functions are called by giving the name of the function followed by the arguments within brackets after the name. For example, to calculate $\sin \pi/2$ we enter the command:

```
>> sin(pi/2)
ans =
     1
```

Code Example 10: Example of calling a function

Some functions can take more than one argument; in this case, we enter the arguments separated by a comma.

```
>> min(pi, 3)
ans =
     3
```

Code Example 11: Example of calling a function with multiple arguments

You can store the results into a variable as normal. Some functions are able to return more than one result, which we shall see in action in Section 3.4.2.

Below is a non-exhaustive list of basic mathematical functions (for complex and/or real numbers). Enter `help elfun` for a more complete list.

`sin`, `cos`, `tan`, `cot`, `sec`, `csc`	Trigonometric functions
`asin`, `acos`, `atan`, `acot`, `sec`, `csc`	Inverse trigonometric functions
`sinh`, `cosh`, `tanh`, `coth`, `sech`, `csch`	Hyperbolic functions
`asinh`, `acosh`, `atanh`, `acoth`, `asech`, `acsch`	Inverse hyperbolic functions
`abs`	Absolute value $\lvert x \rvert$
`exp`	Exponential function e^x
`log`, `log10`, `log2`	Logarithmic function (base e, 10 and 2)
`fix`, `floor`, `ceil`, `round`	Round: to zero, down, up, nearest integer.
`sqrt`, `nthroot`	Square and nth root.
`angle`	Phase angle of a complex number
`conj`	Complex conjugate of a complex number
`real`, `imag`	Real/imaginary parts of complex number

3 Vectors & Matrices

So far we have only dealt with scalar values; however, MATLAB supports matrices and vectors as well. In this section we shall discuss the basics of the matrix support in MATLAB.

3.1 Defining Matrices & Vectors

The basic method to create a vector or matrix in MATLAB is to use square brackets `[]` containing a list of numbers to place in the matrix. Each row of a matrix is a list of numbers separated by either a space and/or a comma, and each row is separated by a semi-colon `;`. For example, the matrix, row vector and column vector,

$$A = \begin{pmatrix} 1 & 9 & 7 \\ -3 & 8 & 0 \\ 2 & -7 & -9 \end{pmatrix}, \qquad x = \begin{pmatrix} 5 & -8 & 0 \end{pmatrix}, \qquad \text{and} \qquad y = \begin{pmatrix} -2 \\ 3 \\ 6 \end{pmatrix},$$

respectively, are generated with:

```
>> A = [1 9 7; -3 8 0; 2 -7 -9]
A =
     1     9     7
    -3     8     0
     2    -7    -9

>> x = [5 -8 0 9]
x =
     5    -8     0     9

>> y = [-2; 3; 6]
y =
    -2
     3
     6
```

Code Example 12: Generating matrices/vectors directly

This notation is really a concatenation of matrices/vectors. The space/comma concatenates columns and the semi-colon concentrates rows. It is, therefore, possible to concatenate matrices into larger matrices using this notation, providing the sizes are compatible.

```
>> B = [A y; x]
B =
     1     9     7    -2
    -3     8     0     3
     2    -7    -9     6
     5    -8     0     9
```

Code Example 13: Matrix/vector concatenation

You can also generate a row vector of a sequence of numbers using the ***start*:*step*:*end*** or ***start*:*end*** syntax, where ***start*** is the first number in the sequence, ***step*** is the difference between elements in the sequence (in the form without ***step*** this defaults to 1), and ***end*** is the largest number (for a positive step) or smallest number (for a negative step) that can be contained in the sequence.

```
>> 1:4
ans =
     1     2     3     4

>> 1:0.5:3
ans =
    1.0000    1.5000    2.0000    2.5000    3.0000

>> 1:2:6
ans =
     1     3     5

>> 1:-1:6
ans =
   Empty matrix: 1-by-0

>> 7:-1:1
ans =
     7     6     5     4     3     2     1

>> 0:0.2:1
ans =
         0    0.2000    0.4000    0.6000    0.8000    1.0000
```

Code Example 14: Generating a vector containing a sequence

As can be seen the ***end*** value is not always included in the sequence. Generating a sequence of equally distributed values, which includes both the start and end values can be done with the

`linspace(start, end, no_points)` function. Here, *`no_points`* is the number of items in the row vector, including the *`start`* and *`end`*.

```
>> linspace(1,2,6)
ans =
    1.0000    1.2000    1.4000    1.6000    1.8000    2.0000

>> linspace(1,0,6)
ans =
    1.0000    0.8000    0.6000    0.4000    0.2000         0
```

Code Example 15: Example usage of `linspace`

MATLAB has several basic functions for generating matrices. The `eye` function generates an identity matrix (ones on the diagonal, zero elsewhere), the `zeros` function generates a matrix of zeros, and the `ones` function generates a matrix of ones. All three functions can take a single scalar argument (N), in which case a $N \times N$ matrix is generated, or two scalar arguments (N and M), in which case a $N \times M$ matrix is generated. These functions can also take a single vector argument (where the vector contains two values $[N\, M]$), which also generates a $N \times M$ matrix. The `rand` and `randn` functions generate a matrix of uniformly or normally distributed random numbers, respectively, between 0 and 1.

```
>> eye(4)
ans =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1

>> zeros(3)
ans =
     0     0     0
     0     0     0
     0     0     0

>> ones(4)
ans =
     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1

>> ones(4,3)
ans =
     1     1     1
     1     1     1
     1     1     1
     1     1     1
```

Code Example 16: Example usage of matrix construction functions

A diagonal matrix can be generated with the `diag` function. This function takes a vector and places the entries on the leading diagonal of a matrix. A second optional scalar integer argument can also be specified, which allows the vector to be placed on a different diagonal. A value of 0 indicates the leading diagonal, a positive number indicates a diagonal above the leading diagonal (with 1 being the diagonal immediately above the leading diagonal) and a negative number indicates a diagonal below the leading diagonal.

```
>> diag([1 2 3 4])
ans =
     1     0     0     0
     0     2     0     0
     0     0     3     0
     0     0     0     4
```

```
>> diag([1 2 3 4],1)
ans =
     0     1     0     0     0
     0     0     2     0     0
     0     0     0     3     0
     0     0     0     0     4
     0     0     0     0     0

>> diag([1 2 3 4],-1)
ans =
     0     0     0     0     0
     1     0     0     0     0
     0     2     0     0     0
     0     0     3     0     0
     0     0     0     4     0
```

Code Example 17: Example usage of `diag`

You can create a complex vector/matrix from two real vector/matrices (representing the real and imaginary parts) by using the **`complex(real,imag)`** function, where ***`real`*** and ***`imag`*** are real matrices representing the real and imaginary parts.

```
>> complex(ones(3),eye(3))
ans =
   1.0000 + 1.0000i   1.0000 + 0.0000i   1.0000 + 0.0000i
   1.0000 + 0.0000i   1.0000 + 1.0000i   1.0000 + 0.0000i
   1.0000 + 0.0000i   1.0000 + 0.0000i   1.0000 + 1.0000i
```

Code Example 18: Generating a complex matrix

Matrices can be *reshaped* into a matrix of a different size using the `reshape` function. This function takes a matrix to reshape as the first argument followed by a matrix size (as either a single vector of two elements or as two arguments). In the form where the new size is specified as two argument you can use an empty vector `[]` to allow MATLAB to automatically calculate the size of that dimension. Note that the number of elements in the reshaped matrix must be the same as in the original matrix. When reshaping a matrix MATLAB works down columns first when reading the elements (and places them in the new matrix in the same way).

```
>> A = rand(4,2)
A =
    0.7363    0.4423
    0.3947    0.0196
    0.6834    0.3309
    0.7040    0.4243

>> reshape(A,[3,3])
Error using reshape
To RESHAPE the number of elements must not change.

>> reshape(A,[2 4])
ans =
    0.7363    0.6834    0.4423    0.3309
    0.3947    0.7040    0.0196    0.4243

>> reshape(A,3,[])
Error using reshape
Product of known dimensions, 3, not divisible into total number of elements, 8.

>> reshape(A,2,[])
ans =
    0.7363    0.6834    0.4423    0.3309
    0.3947    0.7040    0.0196    0.4243
```

Code Example 19: Using `reshape`

3.2 Indexing

Each element in a vector can referred to by using an index notation starting from 1 for the first value (*note that this is different to some other programming languages which start from* 0). In order to access an item of the vector you place the index to access within brackets after the variable name. Matrices require two indices, the first is the row index and the second is the column index (It is possible to use only one index for matrices, in this case the index counts down the first column, then the second, etc.). A special value of `end` can be used to index the last value.

```
>> A = rand(4)
A =
    0.8147    0.6324    0.9575    0.9572
    0.9058    0.0975    0.9649    0.4854
    0.1270    0.2785    0.1576    0.8003
    0.9134    0.5469    0.9706    0.1419

>> A(5)
ans =
    0.6324

>> A(2,3)
ans =
    0.9649

>> x = 1:5
x =
     1     2     3     4     5

>> x(3)
ans =
     3

>> x(end)
ans =
     5
```

Code Example 20: Basic matrix/vector indexing

For an index you can also specify a vector of indices, in which case the result is a vector containing just those values. This vector index can be specified as a normal index or using the ***start*:*step*:*end*** / ***start*:*end*** notation. The special `end` value can be used in this range specifier.

```
>> x([1 3])
ans =
     1     3

>> A(2:3,2:3)
ans =
    0.0975    0.9649
    0.2785    0.1576

>> A(2,2:end)
ans =
    0.0975    0.9649    0.4854

>> A(2,end:-1:2)
ans =
    0.4854    0.9649    0.0975
```

Code Example 21: Vector indices for matrix/vector indexing

You can also use a single colon `:` in an index location to indicate *all* values.

```
>> A(2,:)
ans =
    0.9058    0.0975    0.9649    0.4854
```

```
>> A(:,3)
ans =
    0.9575
    0.9649
    0.1576
    0.9706
```

Code Example 22: Extracting a complete row or column from a matrix

You can also change individual elements in a matrix by indexing as above and using as the left-hand side of the assignment operator `=`. When indexing more than one element the value you assign must either by a scalar (value is assigned to all elements) or a vector/matrix of the same size as the indexed sub-matrix.

```
>> A(2,3) = 1
A =
    0.8147    0.6324    0.9575    0.9572
    0.9058    0.0975    1.0000    0.4854
    0.1270    0.2785    0.1576    0.8003
    0.9134    0.5469    0.9706    0.1419

>> A(end,2:3) = 0.5
A =
    0.8147    0.6324    0.9575    0.9572
    0.9058    0.0975    1.0000    0.4854
    0.1270    0.2785    0.1576    0.8003
    0.9134    0.5000    0.5000    0.1419

>> A(end,2:3) = [0.25 0.2 1]
Subscripted assignment dimension mismatch.

>> A(end,2:3) = [0.25 0.7]
A =
    0.8147    0.6324    0.9575    0.9572
    0.9058    0.0975    1.0000    0.4854
    0.1270    0.2785    0.1576    0.8003
    0.9134    0.2500    0.7000    0.1419
```

Code Example 23: Changing values in a matrix

You can also delete values from a vector/matrix by assigning to them the empty matrix (the brackets `[]` with no contents). In the case of a matrix you can only delete complete rows or columns.

```
>> A(1,:) = []
A =
    0.9058    0.0975    1.0000    0.4854
    0.1270    0.2785    0.1576    0.8003
    0.9134    0.2500    0.7000    0.1419

>> x(2:3) = []
x =
     1     4     5
```

Code Example 24: Deleting vector and matrix entries

3.3 Vector & Matrix Operations

We shall now discuss the basic operations that be used on matrices or vectors.

3.3.1 Matrix/Vector Size

MATLAB has a built-in function `size` which returns a vector containing the dimension of a matrix. The first value in the vector is the number of rows in the matrix, and the second is the

number of columns. When run on a vector one of these values will be 1 (dependant on if the vector is a column or row vector). MATLAB also has a `length` function, which returns a single value denoting the size of the largest dimension of the matrix. This function is most useful for vectors, as it returns the length of the vector.

```
>> size(A)
ans =
     3     4

>> size(x)
ans =
     1     3

>> length(A)
ans =
     4

>> length(x)
ans =
     3
```

Code Example 25: Obtaining matrix/vector size

Note that as `size` returns a two-value vector you can use the result as an argument to the matrix construction functions (see Section 3.1) to construct a matrix the same size as an existing matrix.

3.3.2 Basic Arithmetic

Addition `+` and `-` are defined as normal for vectors and matrices. It is important to note that an error will occur if you try to apply these operators to matrix/vectors of different size. You can, however, apply these operators to a scalar and a vector/matrix. In this case the scalar is automatically converted into a matrix/vector of the correct size filled with the scalar value

```
>> B = rand(size(A))
B =
    0.4218    0.9595    0.8491    0.7577
    0.9157    0.6557    0.9340    0.7431
    0.7922    0.0357    0.6787    0.3922

>> A+B
ans =
    1.3276    1.0570    1.8491    1.2431
    1.0427    0.9342    1.0916    1.5434
    1.7056    0.2857    1.3787    0.5341

>> A+1
ans =
    1.9058    1.0975    2.0000    1.4854
    1.1270    1.2785    1.1576    1.8003
    1.9134    1.2500    1.7000    1.1419
```

Code Example 26: Basic arithmetic with matrices

The multiplication operator `*` can be used to multiply every value of a matrix/vector by a scalar value, or to perform matrix multiplication (providing the two matrices are of compatible sizes). Note that the sequence vector notation generates *row vectors*, so these may need to be transposed (Section 3.3.4).

```
>> A*2
ans =
    1.8116    0.1951    2.0000    0.9708
    0.2540    0.5570    0.3152    1.6006
    1.8268    0.5000    1.4000    0.2838
```

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

>> C = rand(4,3)
C =
    0.6555    0.2769    0.6948
    0.1712    0.0462    0.3171
    0.7060    0.0971    0.9502
    0.0318    0.8235    0.0344

>> A*C
ans =
    1.3319    0.7522    1.6272
    0.2677    0.7223    0.3539
    1.1402    0.4493    1.3840

>> z = [1;2;3;4]
z =
     1
     2
     3
     4

>> A*z
ans =
    6.0424
    4.3579
    4.0809
```

Code Example 27: Matrix multiplication

The division `/` and left division `\` operators can be used to divide each element of a matrix by a scalar (see Section 3.3.5 for another use of these operators). The power operator `^` can be used to raise a matrix by a scalar value (we shall only worry about positive integers here — in which case this operator is equivalent to repeated matrix multiplication of the matrix with itself)

3.3.3 Element-wise Arithmetic

MATLAB also supports element-wise arithmetic operators. When applied to two matrices/vectors of the same size they apply the matching scalar operation to the elements with the same index. If only one of the arguments of the element-wise operator is a scalar then the scalar is automatically converted into a matrix/vector of the correct size filled with the scalar value. The element-wise operators are multiplication (`.*`), division (`./`), left division (`.\`) and power (`.^`). Notice that these are similar to the scalar operators but prefixed with a period.

```
>> A.*B
ans =
    0.3820    0.0936    0.8491    0.3678
    0.1163    0.1826    0.1472    0.5947
    0.7236    0.0089    0.4751    0.0557

>> A./B
ans =
    2.1476    0.1017    1.1777    0.6406
    0.1387    0.4247    0.1688    1.0769
    1.1530    7.0005    1.0313    0.3617

>> A.^B
ans =
    0.9591    0.1072    1.0000    0.5783
    0.1511    0.4325    0.1781    0.8474
    0.9307    0.9517    0.7850    0.4649
```

Code Example 28: Element-wise arithmetic

3.3.4 Transpose

MATLAB has two matrix suffix operators (and matching built-in functions) to take the transpose of a matrix. The *complex conjugate transpose* operator (single-quote `'` or the `ctranspose` function) takes the transpose of a matrix and the complex conjugate in one operation; whereas, the *transpose* operator (`.'` or the `transpose` function) just takes the transpose. For real-valued matrices these two operators are equivalent.

```
>> A'
ans =
    0.9058    0.1270    0.9134
    0.0975    0.2785    0.2500
    1.0000    0.1576    0.7000
    0.4854    0.8003    0.1419

>> A.'
ans =
    0.9058    0.1270    0.9134
    0.0975    0.2785    0.2500
    1.0000    0.1576    0.7000
    0.4854    0.8003    0.1419

>> Z = complex(A,B)
Z =
   0.9058 + 0.4218i   0.0975 + 0.9595i   1.0000 + 0.8491i   0.4854 + 0.7577i
   0.1270 + 0.9157i   0.2785 + 0.6557i   0.1576 + 0.9340i   0.8003 + 0.7431i
   0.9134 + 0.7922i   0.2500 + 0.0357i   0.7000 + 0.6787i   0.1419 + 0.3922i

>> Z'
ans =
   0.9058 - 0.4218i   0.1270 - 0.9157i   0.9134 - 0.7922i
   0.0975 - 0.9595i   0.2785 - 0.6557i   0.2500 - 0.0357i
   1.0000 - 0.8491i   0.1576 - 0.9340i   0.7000 - 0.6787i
   0.4854 - 0.7577i   0.8003 - 0.7431i   0.1419 - 0.3922i

>> Z.'
ans =
   0.9058 + 0.4218i   0.1270 + 0.9157i   0.9134 + 0.7922i
   0.0975 + 0.9595i   0.2785 + 0.6557i   0.2500 + 0.0357i
   1.0000 + 0.8491i   0.1576 + 0.9340i   0.7000 + 0.6787i
   0.4854 + 0.7577i   0.8003 + 0.7431i   0.1419 + 0.3922i
```

Code Example 29: Matrix transposition

3.3.5 Solving Linear Systems

MATLAB has built-in support for solving linear systems by use of the left division (`\`) and division (`/`) operators. Given two matrices A, B and a vector of unknowns x; then, `x = A\B` gives the solution to the equation $Ax = B$ and `x = B/A` gives the solution to the equation $xA = B$. Note that in the first case A and B require the same number of rows and in the second case A and B require the same number of columns.

```
>> A = [3 1 -1; 1 1 1; 0 1 -1]
A =
     3     1    -1
     1     1     1
     0     1    -1

>> B = [0;0;1]
B =
     0
     0
     1
```

```
>> A\B
ans =
   -0.3333
    0.6667
   -0.3333

>> B'/A
ans =
   -0.1667    0.5000   -0.3333
```

Code Example 30: Solving linear systems

3.4 Functions

All basic mathematical functions in Section 2.5 can be applied (usually element-wise) to matrices and vectors.

3.4.1 Vector Functions

Below is a non-exhaustive list of functions for vectors. These can also be applied to matrices, in which case each column of the matrix is treated as a different vector by default, returning a row vector of the results.

`min`, `max`	Minimum/maximum value in the vector
`sum`	Sum of all values
`prod`	Product of all values
`mean`, `median`	Mean/median of the values
`std`, `var`	Standard deviation/variance of the values
`cumsum`	Cumulative sum of the values
`cumprod`	Cumulative product of the values
`sort`	Sorts the values in the vector

3.4.2 Matrix Functions

Below is a non-exhaustive list of functions for matrices. All basic mathematical functions in Section 2.5 can also be applied (usually element-wise) to the matrix.

`inv`	Inverse a matrix (do not use for solving linear systems, see Section 3.3.5)
`det`	Calculate the determinant of a matrix
`trace`	Calculate the trace of a matrix
`norm`	Calculate a norm of the matrix (defaults to 2-norm)
`rank`	Calculate the rank of the matrix
`eig`	Calculate eigenvalues and eigenvectors of the matrix
`poly`	Calculate characteristic polynomial of the matrix
`cond`	Calculate the condition number of the matrix
`expm`, `logm`	Matrix exponential and logarithm
`sqrtm`	Square root of the matrix

Some functions in MATLAB return more than one value. One such example is `eig`. By default a multiple function will return a single result (sometimes the first result only). In the case of `eig` this will be a column vector containing the eigenvalues of the matrix. In order to obtain all the results from the function you need to assign the result directly to multiple variables. This is done by listing the variable names separated by commas and surrounded by square brackets `[]` on the left-hand side of the assignment. In the case of `eig` the two return value version returns a matrix containing each eigenvector as a column and a diagonal matrix of the eigenvalues. If you want to ignore a return value you can use the tilde `~` instead of a function name for that return value.

```
>> eig(A)
ans =
    3.3615
    1.1674
   -1.5289

>> [V,D] = eig(A)
V =
   -0.9011    0.2579    0.2860
   -0.4226   -0.8773   -0.4480
   -0.0969   -0.4048    0.8471

D =
    3.3615         0         0
         0    1.1674         0
         0         0   -1.5289

>> [V,~] = eig(A)
V =
   -0.9011    0.2579    0.2860
   -0.4226   -0.8773   -0.4480
   -0.0969   -0.4048    0.8471
```

Code Example 31: Handling multiple function returns (calculating eigenvalues/eigenvectors)

4 Strings

MATLAB supports string/character values. A string is essentially a special vector of characters, which can be entered using single quotation marks. As a vector you can access sub-strings and individual characters by using standard vector indexing.

```
>> str = 'This is a test string'
str =
This is a test string

>> str(6)
ans =
i

>> str(11:14)
ans =
test
```

Code Example 32: String example

You can concatenate strings together by surrounding multiple string values/literals with square brackets `[]`.

```
>> newstr = ['Concatenate "' str '" in the middle of this string']
newstr =
Concatenate "This is a test string" in the middle of this string
```

Code Example 33: String concatenation

4.1 Formatting Numbers

It is possible to convert numbers into strings using the built-in functions `num2str` and `sprintf`. The first function takes a scalar, vector or matrix and converts it into a string (or array of strings) containing the numbers formatted with at most 4 decimal places. An optional second argument can take a scalar number to specify the number of decimal places to use, or a format argument (see `sprintf`). The `sprintf` function is more complicated (and is based on the function with the

same name from the C programming language). This function takes as the first argument a string which can contain a format specifier (a `'%'` followed by some parameters; for example, `'%08d'` formats an integer padded to 8 characters with leading zeros). For each format specifier a value is read from the rest of the specified arguments (with matrices and vectors being expanded to a list of arguments). The format is applied as many times as necessary to handle all the arguments specified. The MATLAB documentation for this function should be read as it contains far more detail than can be covered here.

```
>> num2str(2.53380112)
ans =
2.5338

>> num2str(2.53380112,7)
ans =
2.533801

>> sprintf('%08d',4)
ans =
00000004

>> sprintf('Test %d %d %d; ', [3 1 -1; 1 1 1; 0 1 -1])
ans =
Test 3 1 0; Test 1 1 1; Test -1 1 -1;
```

Code Example 34: Formatting numbers

Notice that `sprintf` evaluates a matrix column-wise (the first three values printed are the values from the first column).

4.2 Displaying Text

MATLAB by default has outputted the results of its computation in its own format. Using number formatting and strings it is possible to generate customised text output using the `disp` command, which takes a string and outputs it to the *Command Window*.

```
>> x = 1:10;
>> disp(['First 10 factorials:' sprintf('\n%4d: %8d', [x; cumprod(x)])])
First 10 factorials:
   1:        1
   2:        2
   3:        6
   4:       24
   5:      120
   6:      720
   7:     5040
   8:    40320
   9:   362880
  10:  3628800
```

Code Example 35: Displaying text

Notice in the `sprintf` function a `'\n'` in the string inserts a new line in the output. A `fprintf` function also exists, which works in a similar manner to `sprintf`, but rather than return a string it prints the result directly to the *Command Window* without a new line at the end.

5 Graphics

MATLAB has support for plotting data in various formats. In this section we shall discuss the basic plotting functions available.

5.1 Plot Basics

The main basic plotting function is simply called `plot`. This function is used to plot x and y data against each other as a line plot. The function can take a variable number of arguments, with each set of three arguments (a triple) detailing a set of data to plot, and how to plot it. The first argument is the x data for plot, the second argument is the y data and the third argument is a string *line specification* specifying the format for the line. A line specification consists of three parts: a colour, a marker and a line type.

Colour		Marker		Line	
`b`	Blue	`.`	Point	`-`	Solid
`g`	Green	`o`	Circle	`:`	Dotted
`r`	Red	`x`	Cross	`-.`	Dash-dot
`c`	Cyan	`+`	Plus	`--`	Dashed
`m`	Magenta	`*`	Star	(none)	No line
`y`	Yellow	`s`	Square		
`k`	Black	`d`	Diamond		
`w`	White	`v`	Triangle (down)		
		`^`	Triangle (up)		
		`<`	Triangle (left)		
		`>`	Triangle (right)		
		`p`	Pentagram		
		`h`	Hexagram		

After each triple of data we can also specify key-value pairs, where the key is a string, for specifying more detailed plot information (use `doc plot` for more information). The `plot` function can also take a single vector argument, in which each item in the vector is plotted against the vector index, or just two vector arguments (the x and y values). In these case a default *line specification* is used.

```
>> x = 0:0.5:3
x =
         0    0.5000    1.0000    1.5000    2.0000    2.5000    3.0000

>> plot(x,x,'-xr',x,x.^2,'-ob',x,x.^3,'-sk')
```

Code Example 36: Basic plotting

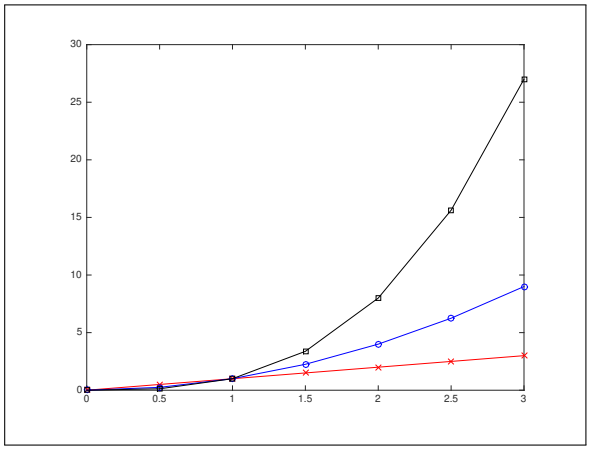

Figure 2: Basic plotting result

Here, MATLAB has automatically decided the minimum and maximum values to display on the x and y axis. Three functions exist for changing these limits. `xlim([min max])` sets the limits of

the x axis, `ylim([min max])` sets the limits of the y axis, and `axis([xmin xmax ymin ymax])` sets the limits of both axis. All these functions are applied to the currently active plot.

`plot` uses a linear x and y axis. There also exists the functions `loglog`, `semilogx`, and `semilogy`, which use logarithmic xy, logarithmic x/linear y, and linear x/logarithmic y, respectively.

5.2 Annotation

In the previous example of plotting you will notice that there is no title or axis labels. We can add these by use of the `title`, `xlabel` and `ylabel` commands. Note that basic TEX support is implemented in the string specified in the arguments (full LATEX support can also be enabled). We would also like to be able to add a legend to the plot. We do this with a the `legend` function that takes a list of strings as the legend for each plot (the first string is the first plot, etc.). The legend can also take key-value pairs for more detailed settings. The most important of these is the `'Location'` value, which allows us to position the legend on the plot.

```
>> xlabel('x')
>> ylabel('f(x)')
>> title('Basic Polynomial Functions')
>> legend('x','x^2','x^3','Location','NorthWest')
```

Code Example 37: Annotating a plot

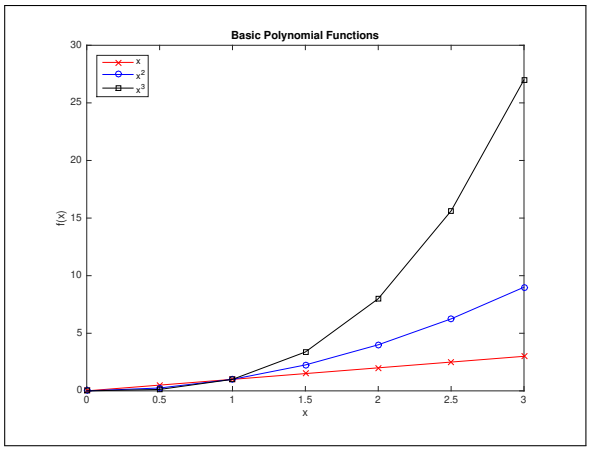

Figure 3: Annotated plotting result

As can be seen here, the font can be a little small, this can be changed with key-value pair arguments to all the annotation functions. However, often a simpler way is to use the interactive *Plot Tools*. To open this on an active plot, either enter `plottools` at the prompt, or use the Show Plot Tools and Dock Figure tool-bar button ( ) on the plot window. You can see the MATLAB code required to generate a plot by clicking the File > Generate Code... menu item.

5.3 3D Plots

MATLAB also has a number of functions for plotting three-dimensional data (x, y and z data). The first we shall consider is the `plot3` function, used to plot a 3D line plot. This function works in a similar manner to `plot`, except you need to provide vectors of x, y and z data for each line.

```
>> t = linspace(0,10*pi,501);
>> plot3(sin(t),cos(t),t,'-r')
```

Code Example 38: 3D line (parametric) plot

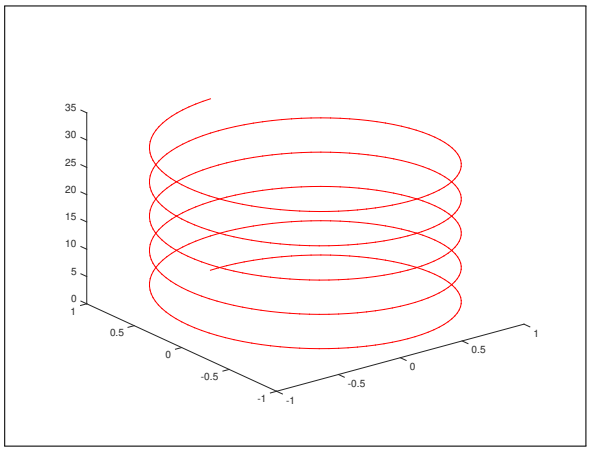

Figure 4: 3D line (parametric) plot

The annotations work as before, although we also have a `zlabel` function for labelling the z axis.

We can also plot full 3D functions using the `surf` plot. The normal usage of this function takes three arguments, for the x, y and z data. The z data must be a $M \times N$ matrix; whereas, x and y can be a matrix or a vector (but both must be the same). If x and y are matrices they must be the same size as the z matrix and each matching element from the three vectors denotes a x, y and z point to plot on the surface. If x and y are vectors then x must be of length N and y must be of length M; in this case, each point plotted is $(x(j), y(i), z(i, j)$. MATLAB has an in-built function called `meshgrid`, which takes a x and y vector and creates a full x and y matrix representing the tensor points.

```
>> x = linspace(0,1,5);
>> [X, Y] = meshgrid(x,x)
X =
         0    0.2500    0.5000    0.7500    1.0000
         0    0.2500    0.5000    0.7500    1.0000
         0    0.2500    0.5000    0.7500    1.0000
         0    0.2500    0.5000    0.7500    1.0000
         0    0.2500    0.5000    0.7500    1.0000

Y =
         0         0         0         0         0
    0.2500    0.2500    0.2500    0.2500    0.2500
    0.5000    0.5000    0.5000    0.5000    0.5000
    0.7500    0.7500    0.7500    0.7500    0.7500
    1.0000    1.0000    1.0000    1.0000    1.0000

>> x = linspace(0,1,25);
>> [X, Y] = meshgrid(x,x);
>> surf(X, Y, X.*Y.*(1-X).*(1-Y))
>> colorbar
```

Code Example 39: 3D surface plot

We have used the `colorbar` function to display a colour bar on the active plot. We can change the colour scheme used by the `colormap` function, which either takes a string specifying a built-in colour map or a $N \times 3$ matrix of colours (each row is a Red-Green-Blue colour triplet, each value between 0 and 1). See `help graph3d` for a list available colour maps.

The Rotate 3D tool-bar button ( ) in the plot window allows you to rotate the plot by dragging within the plot area. The `view(2)` or `view(3)` function calls will switch the current plot view to a 2D top-down view or the default 3D view, respectively.

The x and y arguments to `surf` can be omitted, in which case the values in z are plotted against their indices. The colour of the surface plot is by default calculated from the z value. Optionally, a

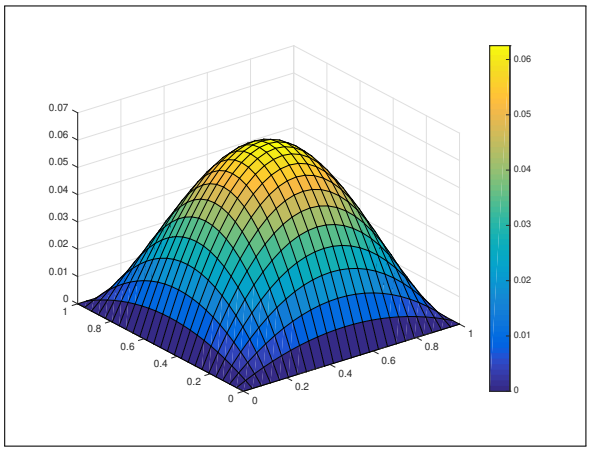

Figure 5: 3D surface plot

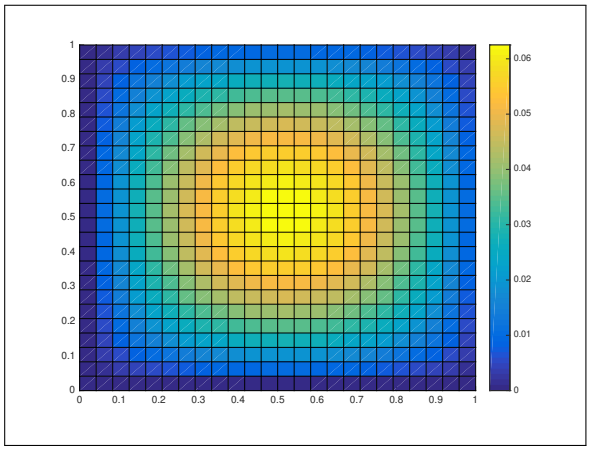

Figure 6: Result of `view(2)` on 3D surface plot

fourth *colour* argument can be specified, which specifies the colour (the colour argument of each point is a single value and MATLAB interpolates the colour in the same way as when using z data).

MATLAB contains several more useful functions for plotting 3D data. The `mesh` function is similar to `surf`, except it only plots the mesh lines (this time coloured), rather than filled polygons. The `trisurf` and `trimesh` functions take only vector x, y, and z data, but also takes as the first argument a $M \times 3$ matrix of *triangles*; this matrix denotes in each row the indices (in the x, y and z data) of a single triangle to plot. The `contour` function is similar to `surf`, but plots a 2D contour plot; an optional fourth argument can be used to specify the number of contour levels. Finally, `surfc` and `meshc` draws a surface or mesh plot, respectively, with a contour plot underneath.

```
>> x = linspace(-pi,pi,25);
>> [X, Y] = meshgrid(x,x);
>> meshc(X, Y, sin(X).*sin(Y))
```

Code Example 40: 3D mesh/contour plot

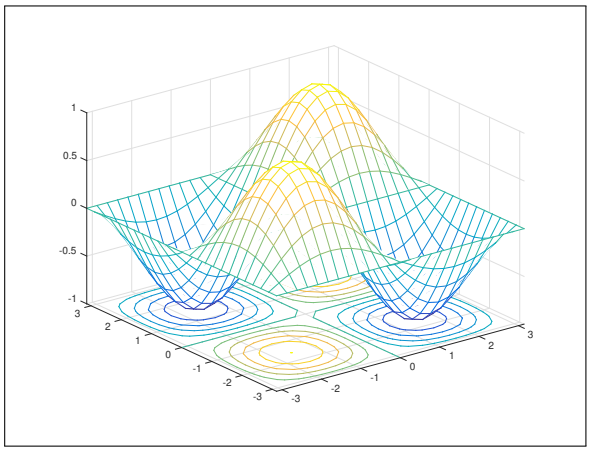

Figure 7: 3D mesh/contour plot

6 Programming

So far we have only run MATLAB commands one at a time within the *Command Window*. While this is useful for testing and learning proposes we need to be able to run multiple commands as a single program.

6.1 Scripts

The simplest method for running multiple commands is the use of MATLAB scripts. A MATLAB script is basically a simple text file (with a `.m` extension) filled with MATLAB commands to execute. Scripts are executed by typing the name of the file (without extension) at the prompt. The script is found by looking in the path and current folder (as described in Section 1.2). Note that the current folder is searched first, so a file in the local folder with the same name as a built-in MATLAB function will overwrite the MATLAB function (so a script file called `sin` will result in all calls to the sine function producing incorrect results). All variables and values in the script file are saved into the workspace, so the values can be accessed after execution, and the script can access any variables currently in the workspace.

In order to start editing a script file within MATLAB you can either create a new file with Home > New Script or Home > New > Script menu bar buttons, or you can type `edit filename` into the *Command Window* (where `filename` is the name of the file with or without extension). In the first case the file will not be created until you save it. In the later case, if a script with the specified name already exists it will be opened for editing; otherwise, a prompt will ask if you want to create a script with that name. In either case the *Editor* window will appear with a blank text file, simple enter the script and save.

MATLAB scripts may contain comments (lines of text which are not executed) and blank lines. Comments in MATLAB start with a percentage sign `%` and continue until the end of the line. When run the script will output the result of any command not ending in a semi-colon `;` (the same as if run from the *Command Window*); therefore, it is advisable in scripts to always end lines with a semi-colon. With scripts it is desirable to keep lines short (so they can be easily read); in fact, the MATLAB *Editor* window includes a line down the page to indicate a sensible line length (comments are automatically split at this point). If you enter a long command and need to split it over multiple lines you need to use the *line continuation ellipsis* `...` (three periods) at the end of the line to continue. Note that you cannot use this in the middle a string (split the string and use string concatenation, see Section 4).

Below is a sample script to demonstrate a few of these point, note the comments and the `fprintf` function call which is split over two lines. Create a script file `sample_script.m` in a local

directory (you may want a separate directory for this course) containing the following code.

```
% This is a comment

% Set up variables
A = [3 1 -1; 1 1 1; 0 1 -1];
B = [0;0;1];

% Print out some matrix properties
disp('Properties of A:');
disp(['    Condition number:          ' num2str(cond(A))]);
disp(['    Determinant:               ' num2str(det(A))]);
% Following command is split over two lines
fprintf('    Characteristic Polynomial: %.1fx^3%+.1fx^2%+.1fx%+.1f\n\n', ...
    charpoly(A));

% Solve Ax=b
disp('Solving Ax=b');
x = A\B

% Display the output
disp(['x = [' sprintf('%8.4f', x) ']']);
```

Code Example 41: Sample script

To execute the script, do one of the two following options:

- Ensure you are in the directory containing the file (`cd` as necessary) and enter **`sample_script`** into the *Command Window*, or
- Click the Run button in the Editor tab of the *Editor* window with the script file open (if you are not in the correct folder MATLAB will present a warning dialog — select *Change Folder* from this dialog).

Either method will run the code and output the results.

```
>> sample_script
Properties of A:
    Condition number:          3.2355
    Determinant:               -6
    Characteristic Polynomial: 1.0x^3-3.0x^2-3.0x+6.0

Solving Ax=b
x = [ -0.3333   0.6667  -0.3333]
```

Code Example 42: Sample script output

The *Editor* underlines warnings (in *orange*) and errors (in *red*) within a script file, with a matching symbol in the right margin. Hovering over these with the cursor will display a message on the problem, and a button to potentially fix the problem (if MATLAB can calculate an obvious fix). Errors are problems that will stop the script running (for example forgetting the ... when splitting over two lines the command in the above example). Warnings are things that MATLAB considers bad practice (forgetting a semi-colon at the end of an assignment), but will still allow the code to be run.

6.2 Functions

So far we have only used built-in MATLAB functions. MATLAB as a programming language allows us to define our own functions as necessary. These are defined in a script file similar to basic scripts, but with a specific format (a required header line defining the function). You can create a function by editing a script file as specified above and entering the necessary code, or to have MATLAB automatically generate a template use the Home > New > Function menu bar button.

```
function [ output_args ] = functionname( input_args )
%FUNCTIONNAME Summary of this function goes here
%   Detailed explanation goes here

end
```

Code Example 43: Default function structure

The above shows an example of an automatically generated function template. This will need editing as required. The definition of the function breaks down into the following components:

`function` **Keyword** Keyword to denote we are writing a function.

output_args Comma-separated list of output argument names. If the function has no output then the code between the `function` keyword and the ***functionname*** can be omitted (`[ output_args ] =`).

functionname The name of the function (used when calling the function). The file name of the file containing the function must be ***functionname*.m** (MATLAB will display a warning if the file name and function name differ).

input_args Comma-separated list of input argument names.

H1 Comment Line The first comment line immediately following the function definition line. Should contain the function name followed by a very short description.

Further documentation comments More comment lines immediately following the function definition. The text in these comment lines, along with the *H1 Comment Line*, will be displayed when `doc` ***functionname*** or `help` ***functionname*** are called.

Main Body All code between the function definition and the matching `end` keyword. Makes up the code executed when the function is called

`end` **Keyword** The end of the function definition. Note this is optional if this is the only function defined within the file.

Functions variables have scope only to the function. This means that the function does not have access to any variable in the workspace, and any variables set in the function are not available in the workspace. To allow arguments to be passed to and from your function you specify a list of input arguments in the brackets after the function name and a list of output arguments in the square brackets after the `function` keyword. The input arguments will have the values of whatever is passed when the function is called. To return values in the output arguments, simply assign the result to return to the variable name of the output argument. When the function is called the caller specifies how many return values to handle (see Section 3.4.2) by the variables that the result is assigned to. If you function returns more variables then these are ignored. A special variable `nargout` is available to functions, which gives the number of return values the caller requested. You can also allow a variable number of input arguments, but that is beyond the scope of this course.

```
function [ z, w ] = sample_function( x, y )
%SAMPLE_FUNCTION Calculates the product and difference of two numbers
z = x*y;
w = x-y;
end
```

Code Example 44: Simple function

The above sample show the definition of a function with two returns. This function is called like any other MATLAB function (providing it is in the current folder/path).

```
>> x = sample_function(2,3)
x =
     6

>> [x,y] = sample_function(2,3)
x =
     6

y =
    -1
```

Code Example 45: Calling the simple function

There is one problem with the function. While it works for scalar inputs it will fail with vector or matrix based inputs. Often when we write functions we want them to be as generic as possible (notice that `sin`, for example, is defined for scalar, matrix or vector inputs). We can fix this by using the element-wise operators in preference to the normal scalar operators.

```
1  function [ z, w ] = sample_function_vec( x, y )
2  %SAMPLE_FUNCTION_VEC Calculates the product and difference of two inputs
3  z = x.*y;
4  w = x-y;
5  end
```

Code Example 46: Simple function changed for vector/matrix inputs

```
>> [x,y] = sample_function([1 5],[2 5])
Error using  *
Inner matrix dimensions must agree.

Error in sample_function (line 3)
z = x*y;

>> [x,y] = sample_function_vec([1 5],[2 5])
x =
     2    25

y =
    -1     0
```

Code Example 47: Calling functions with vector arguments

Function files can actually contain more than one function; however, only the first "main" function defined in a file is *visible* outside the file. This allows you to write local functions used by the main function in the file but not usable by anyone else.

```
1  function [ z, w ] = sample_subfunction( x, y )
2  %SAMPLE_SUBFUNCTION Calculates the product and difference of two inputs
3  z = multiply(x, y);
4  w = x-y;
5  end
6
7  function [z] = multiply(x,y)
8  %MULTIPLY Elementwise multiplication of the vectors
9  z = x.*y;
10 end
```

Code Example 48: Sub-function example

In this case the `multiply` is only available to the `sample_subfunction` function. Note that we can still have comments after the function definition of a sub-function. To see the documentation you need to specify both the main function name and the sub-function name separated by a `>` in the call to `help` or `doc`. For example, to see the documentation of the `multiply` sub-function simply call `help sample_subfunction>multiply`.

A special statement, `return`, can be used anywhere in a function. When this statement is executed MATLAB exits the current function and executes no more commands from the function. This can be combined with `if` statements (see Section 6.5) to handle special cases or abort early. *Unlike in some other programming languages* `return` *does not take any arguments (a return value), as return values are set by assignment.*

6.3 Logical Operators

MATLAB has a built-in *logical* type (often called *boolean* type in other language), which can take a *true* or *false* value only (represented in MATLAB by 1 and 0, respectively). This logical type is returned when using any of the MATLAB *comparison operators*. These comparison operators are performed element-wise on matrices of the same size and return a matrix with 1 (true) or 0 (false) in each entry indicating if the condition is true for the matching elements. The comparison operators can be used with one scalar and one vector/matrix argument, in which case the scalar is expanded into a matrix of the correct size.

`A < B`	Checks if $A < B$ (element-wise)
`A > B`	Checks if $A > B$ (element-wise)
`A <= B`	Checks if $A \leq B$ (element-wise)
`A >= B`	Checks if $A \geq B$ (element-wise)
`A == B`	Checks if $A = B$ (element-wise)
`A ~= B`	Checks if $A \neq B$ (element-wise)

The first four of these work only on the real part of a complex number; whereas, `==` and `~=` works on both the real and imaginary parts. In Section 2.2 we mentioned that double-precision floating points suffer from rounding error. The main issue this results in is that the equality comparisons do not always return true when we expect, as although two numbers may appear to be the same there may be a small difference. For example, on my machine, the following occurs:

```
>> (19.2-19) == 0.2
ans =
     0

>> 19.2-19-0.2
ans =
    -7.216449660063518e-16
```

Code Example 49: Example of rounding error in comparisons

Generally, if we are dealing with a comparison of floating point numbers we calculate the absolute value of the difference between the numbers and check if it less than some tolerance value (a small number). For example, rather than evaluate `A==B`, we instead use, `abs(A-B) < 1e-8`. It also worth noting that the special value `NaN` always compares as not equal to any value, including itself (`NaN == NaN` returns *false*). To detect `NaN` values you should use the `isnan` function.

The comparison operators above work on numbers. MATLAB has two functions, `strcmp` and `strcmpi`, which perform *case sensitive* and *case insensitive*, respectively, comparisons (equality) on two strings.

MATLAB has two functions, called `true` and `false`, which work in a similar way to `ones` and `zeros`, respectively, except that the return values are *logical* rather than *double*. These functions can also be used without arguments to return a scalar *true* or *false* value.

MATLAB also has a number of *logical operators* and functions, which act (element-wise) on matrices of logical values.

Operator	Function	Description
`~A`	`not(A)`	Performs a logical NOT (0 becomes 1, 1 becomes 0)
`A & B`	`and(A,B)`	Performs logical AND (returns 1 if both A and B are 1)
`A | B`	`or(A,B)`	Performs logical OR (returns 1 if either A or B are 1)
	`xor(A,B)`	Performs logical XOR (returns 1 if *only one* of A and B are 1)

Two functions (`all` and `any`) exist, which can be applied to a matrix or vector. Applied to a vector these functions perform a logical AND or OR, respectively, to the elements of the vector; when applied to a matrix the logical AND or OR is applied to each column (returning a row vector).

```
>> x = rand(3)
x =
    0.2599    0.1818    0.8693
    0.8001    0.2638    0.5797
    0.4314    0.1455    0.5499

>> y = x > 0.5
y =
     0     0     1
     1     0     1
     0     0     1

>> z = x < 0.2
z =
     0     1     0
     0     0     0
     0     1     0

>> y | z
ans =
     0     1     1
     1     0     1
     0     1     1

>> w = any(y)
w =

     1     0     1

>> all(w)
ans =
     0
```

Code Example 50: Logical operations

Two, *short-circuit* logical operators also exist. The short-circuit logical AND `&&` and short-circuit logical OR `||` can only be applied to scalar arguments. These functions are called short-circuit because they only evaluate the second argument if necessary. Basically, for the short-circuit logical AND the second argument will not be evaluated if the first argument is *false* and for the short-circuit logical OR the second argument will not be evaluated if the first argument is *true* (because in both cases MATLAB already knows the result of the logical operator).

The main use of logical values will arise in the next few sections; however, they can also be used to index into a vector/matrix. If a logical matrix is passed to the index argument of a matrix of the same size it will return only the values whose matching logical element is *true*.

```
>> x = rand(4)
x =
    0.7803    0.0965    0.5752    0.8212
    0.3897    0.1320    0.0598    0.0154
    0.2417    0.9421    0.2348    0.0430
    0.4039    0.9561    0.3532    0.1690

>> x(x < 0.5) = 0
x =
    0.7803         0    0.5752    0.8212
         0         0         0         0
         0    0.9421         0         0
         0    0.9561         0         0
```

Code Example 51: Using logical indexing

A `find` function also exists in MATLAB which returns the indices of all non-zero entries in a matrix or vector. This can be passed a logical matrix to find all entries in a matrix which meet a certain condition.

```
>> [i,j] = find(x < 0.02)
i =
     2

j =
     4
```

Code Example 52: `find` indices of all values < 0.02 in matrix

6.4 Loops

So far we have only considered statements which are executed once. *Loops* are a programming structure that allows us to execute a number of statements several times. MATLAB has both `for` and `while` loop types. *An important note is that although we can use loops to iterate through the elements in a vector/matrix and handle them one at a time this should be avoided wherever possible (by using element-wise and matrix operations) for performance (speed) reasons.*

6.4.1 for Statement

A `for` loop executes a set of statements a set number of times, each time with an *index* variable which takes the next value from an vector/matrix. The general structure of the *for* loop is:

```
1 for index = values
2     statements
3 end
```

Code Example 53: `for` loop structure

Here, ***index*** is a variable name to use as the index variable and ***values*** is the vector/matrix of values to take. The first time the ***statements*** are executed ***index*** will be a column vector containing the first column of ***values***, the second time ***statements*** are executed ***index*** will be a column vector containing the second column of ***values***, etc.. In normal usage ***values*** is a row vector, so each iteration ***index*** takes a single value; in fact, usually we use the ***start:end*** or ***start:step:end*** notation directly.

```
1 disp('The first 10 factorials');
2 v = 1;
3 for i = 1:10
4     v = v*i;
5     fprintf('%3d : %8d\n', i, v);
6 end
```

Code Example 54: Factorial calculation with `for` loop

```
The first 10 factorials
  1 :        1
  2 :        2
  3 :        6
  4 :       24
  5 :      120
  6 :      720
  7 :     5040
  8 :    40320
  9 :   362880
 10 :  3628800
```

Code Example 55: Result of factorial calculation with `for` loop

Notice that the statements within the `for` loop are indented. This is sensible to do with any block-style statement as it makes the code easier to read (the MATLAB editor will actually automatically indent for you, and unindent when you enter `end`).

Within a `for` loop, two special statements can be used. `break` exits the loop completely (no more values from ***values*** list are evaluated) when it is called and `continue` stops executing the ***statements*** for the current value and moves to the next value from ***values*** (if one exists).

6.4.2 while Statement

The `while` loop continues to execute the statements it contains while some condition holds true.

```
1 while expression
2     statements
3 end
```

Code Example 56: `while` loop structure

When this statement is reached the ***expression***, which should return a logical matrix, is evaluated. If all values in the matrix are non-zero (true) then the ***statements*** are executed. The ***expression*** is then evaluated again and ***statements*** executed if all entries of the matrix are non-zero. This continues until one entry of ***expression*** evaluates to 0 (false).

```
1 v = 100;
2 while v > 0.5
3     disp(num2str(v,8));
4     v = v/2;
5 end
```

Code Example 57: `while` loop example

```
100
50
25
12.5
6.25
3.125
1.5625
0.78125
```

Code Example 58: Result of `while` loop example

If the ***expression*** never evaluates to false then the code will be stuck in an *infinite loop* and the script will never terminate. You can force a running script to terminate by using the Ctrl + C keyboard shortcut within the *Command Window*.

Within a `while` loop `break` and `continue` can be used in similar manner to a `for` loop. `break` exits the loop, `continue` stops executing the ***statements*** and evaluates ***expression*** again.

6.5 if-else Statement

Using the logical values we have seen in Section 6.3 is it possible to write MATLAB code which only executes statements if a particular condition is true. To do this we use a `if` statement, which has the general form:

```
1 if expression
2     if_statements
3 elseif expression
4     elseif_statements
5 else
6     else_statements
7 end
```

Code Example 59: `if`-`elseif`-`else` structure

The `elseif` block (with its following statements) and the `else` block (with its following statements) are both optional. You can also have multiple `elseif` statements (must be after the `if` statement and before the `else` statement). When an `if` statement is reached the (logical) ***expression*** is evaluated. If all values in the resulting matrix are non-zero (true) then ***if_statements*** are executed; however, if ***expression*** evaluates to false then the ***expression*** for the first `elseif` statement is evaluated and ***elseif_statements*** executed if this is true. This continues, evaluating the ***expression*** for each `elseif` statement, in order, until one is true (then the matching statements are executed). If no ***expression*** evaluates to true then the ***else_statements*** are executed.

```
function plotdata(x, y, xtype, ytype)
% plotdata Plots the data on linear or log plots
%     xtype and ytype are strings specify the type of
%     scale for that axis - either 'log' or 'linear'.
if (strcmpi(xtype,'log') && strcmpi(ytype,'log'))
    loglog(x, y);
elseif strcmpi(xtype,'log')
    semilogx(x, y);
elseif strcmpi(ytype,'log')
    semilogy(x, y);
else
    plot(x, y);
end
end
```

Code Example 60: `if` example

6.6 switch Statement

The `switch` statement selects a set of statements to execute based on the value of a number or string. The structure of the `switch` statement is as follows.

```
switch expression
    case case_expression
        statements
    otherwise
        otherwise_statements
end
```

Code Example 61: `switch` structure

The `otherwise` statement is optional and you can have multiple `case` statements. When reached a `switch` statement evaluates the ***expression*** and compares the result against all ***case_expression***s. The statements of the first matching ***case_expression*** are executed. If no match occurs then the `otherwise_statements` are evaluated. A `case_expression` can be a single value, or multiple values (comma-separated and surrounded by braces {}) if multiple values should have the same statements executed. *Note that unlike in some other programming languages (C/C++ etc.)* ***case*** *statements do not have fall-through behaviour and MUST NOT contain a* ***break*** *statement.*

```
function [city] = capital(country)
switch country
    case 'Austria'
        city = 'Vienna';
    case 'Germany'
        city = 'Berlin';
    case {'United Kingdom','Great Britain'}
        city = 'London';
    otherwise
        city = '<Unknown>';
end
end
```

Code Example 62: `switch` example

Note that the string comparison here is *case sensitive*.

6.7 Function Handles & Anonymous Functions

So far all variables/values we have considered have been some form of number or a string. It is possible, however, to store a reference to a function (called *function handles* in MATLAB terminology) within a variable. You can then call that function via the variable by just that variable name like a function. To take a handle of a function just use the function name prefixed by a `@` symbol.

```
>> sinhandle = @sin
sinhandle =
    @sin

>> sinhandle(pi/2)
ans =
     1
```

Code Example 63: Taking and using function handle of sin

The main use for this is that it allows generic functions to be written that operate on a function, without having to know what function it operates on. If you write such a function you just treat the specific argument as a function handle and you only need to know how many arguments to pass to the function. (Note that the function comments should document this for other users of your functions). MATLAB has several functions that take function handles, such as the *easy* plotter functions. Almost all plotting functions we discussed in Section 5 have a *easy* version with the same name prefixed by **`ez`**, such as **`ezsurf`**, **`ezplot`**, **`ezcontour`**, etc.. This functions take a function handle (or a string containing MATLAB code to evaluate) and plots the function over the interval $[-2\pi, 2\pi]$ (in each coordinate direction), picking the points it samples the function at itself. You can also specify a vector as a second argument to change the range. The number of arguments the function handle takes depends on the plotting routine (**`ezplot`** takes one argument and **`ezsurf`** takes two arguments).

```
>> ezplot(@sin,[-pi pi]);
```

Code Example 64: Calling **`ezplot`** with function handle of sin

We can use function handles to functions we have written as well. For example we can call **`ezsurf`** on the **`sample_function`** and **`sample_function_vec`** functions we wrote in Section 6.2.

```
>> ezsurf(@sample_function)
Warning: Function failed to evaluate on array inputs; vectorizing the
function may speed up its evaluation and avoid the need to loop over array
elements.
> In ezplotfeval (line 56)
  In ezgraph3>ezeval (line 635)
...
>> ezsurf(@sample_function_vec)
```

Code Example 65: Calling **`ezsurf`** with handle to own function

In Section 6.2 we mentioned that functions should be written as generic as possible, we have here another demonstration of why. The **`ezsurf`** function has tried to call the function with vector or matrix inputs (essentially one call with all points it wants to evaluate), but as that failed in the first call it then fell back to one point at a time.

Using function handles we can also define *anonymous functions*. This are functions that are written inline in MATLAB (usually fairly simple one-line functions). An anonymous function is written as **`@(`*`input_args`*`)`** ***`functioncode`***, where ***`input_args`*** is the same as for a normal function and ***`functioncode`*** is a line of MATLAB code which returns a result (which is the result of the function). You can use this anonymous function like a normal function handle (passing to a function or assigning to a variable). We can, for example, write Code Example 40 as follows.

```
>> ezmeshc(@(x, y) sin(x).*sin(y), [-pi pi])
```

Code Example 66: 3D mesh/contour plot using anonymous functions

7 Structures

MATLAB has support for various types of more complex data structures, which are beyond the scope of this course. One basic complex data type is the **`struct`** type. A structure is essentially a group of variables stored together in a single object. The various variables (fields) in a structure can be different types. A structure type can be generated in two different ways, either with the **`struct`** function, or by direct assignment of fields. The **`struct`** function takes a variable number of values, where each pair is a *key-value* pair (the first value is the key and MUST be a string, the second is the value).

```
>> course = struct('Name','Numerical Solution of ODEs',...
    'Year',2019,'Semester','Winter')
course =
        Name: 'Numerical Solution of ODEs'
        Year: 2019
    Semester: 'Winter'
```

Code Example 67: Generating structure using **`struct`** function

You can access a field, for reading or assignment of value, by using the *dot* . notation. Here, you use the variable name, followed by a period and then the name of the field.

```
>> course.Name = 'Numerical Solution of ODEs';
>> course.Year = 2019;
>> course.Semester = 'Winter';
>> course
course =
        Name: 'Numerical Solution of ODEs'
        Year: 2019
    Semester: 'Winter'

>> course.Name
ans =
Numerical Solution of ODEs
```

Code Example 68: Generating and reading structure directly

You can also create structure arrays. Structure arrays are accessed in the same way as vectors, and assigning fields to an index that doesn't currently exist will grow the array to the correct size.

```
>> course(3).Year = 2018
course =
1x3 struct array with fields:
    Name
    Year
    Semester

>> course(2)
ans =
        Name: []
        Year: []
    Semester: []

>> course(3)
ans =
        Name: []
        Year: 2018
    Semester: []
```

Code Example 69: Accessing structure array

Structures, and structure arrays, can be nested with a structure field containing another structure or array.

8 Error Handling

In some of the code examples above you will have noticed red and orange messages appearing in the *Command Window*. This are errors, and warnings, that occurred while trying to run the code. In this section we shall discuss how to handle errors, and incorrect results.

8.1 Understanding Error Messages

When an error message appears the key is understanding what it is trying to explain. In most cases this is fairly obvious (although occasionally a different error message to the real problem may appear).

```
>> A = rand(4);
>> B = rand(5);
>> A*B
Error using  *
Inner matrix dimensions must agree.

>> sample_function
Error using sample_function (line 3)
Not enough input arguments.
```

Code Example 70: Example error messages

In the case of the first error, we are trying to matrix multiple two matrices with incompatible dimensions (a 4×4 with a 5×5), so MATLAB complains about `Inner matrix dimensions`, as the *inner matrix dimensions* are the number of columns in the first matrix and the number of rows in the second matrix. In the second case we have tried to call a function without the correct number of arguments. As this is a function we wrote it has also given us a line number in the function where the error occurred. This line number is a clickable link, which will open the *Editor* window at the specified line. (In this case the error is not here, but it is the location were it is detected — the line the arguments are first used).

The second situation highlights a useful feature. Take the following code (see if you can spot the error before running).

```
1 function [b] = invalid_func(n)
2 % invalid_func Function that we want to take a number
3 %     and perform Ax for A=rand(n), x=1:n
4 A = rand(n);
5 x = 1:n;
6 b = A*x;
7 end
```

Code Example 71: Function with error

When we try to run this function we get an error.

```
>> invalid_func(4)
Error using  *
Inner matrix dimensions must agree.

Error in invalid_func (line 6)
b = A*x;
```

Code Example 72: Running function with error

MATLAB has told us the error (matrix multiplication with incorrect dimensions), the line the error occurred on, and has even printed the line causing the error as well. So if we click the line number we can then look at the code and try and find the error. The error is a matrix dimension problem so we need to look at the sizes of A and x. `A=rand(n)`, so that is a $n \times n$ matrix, which is as we expect. `x=1:n`, so that is a vector of n values; however, it is a *row vector*, or a $1 \times n$ matrix. So the solution is just to transpose x.

8.2 Generating Errors

When you are writing a code it is possible that you will want to generate error messages in certain cases. The most common of these is checking that input values to a function are valid. When you want to generate an error call the `error` function, passing a error message (string) to display. When this call is executed the function terminates and the error message is displayed in the *Command Window*.

```
function [x] = basic_factorial(n)
% basic_factorial A very basic (and naive) factorial implementation

x = 1;
if (n < 0)
    error('Factorial only defined for non-negative numbers');
elseif (round(n) ~= n)
    error('Factorial only defined for integer values');
elseif (n > 0)
    for i=1:n
        x = x*i;
    end
end

end
```

Code Example 73: Generating errors

```
>> basic_factorial(-2)
Error using basic_factorial (line 6)
Factorial only defined for non-negative numbers

>> basic_factorial(2.3)
Error using basic_factorial (line 8)
Factorial only defined for integer values

>> basic_factorial(0)
ans =
     1

>> basic_factorial(4)
ans =
    24
```

Code Example 74: Executing a function with error checking

The `error` function can also take a string, followed by a variable number of arguments. In this case it treats the string and the arguments in the same way as `sprintf`. A `warning` function also exists that takes identically arguments to `error`. This function generates a warning message in the *Command Window* when executed, but allows the script to continue running.

8.3 Debugging

When running scripts or functions it is possible an error occurs you were not expecting, or an unexpected value is returned. MATLAB has a debugger, which allows you to run the code and inspect the values within a function. The best way to enter debug mode is to place a *breakpoint* on a line of code where you want the execution to pause and then execute the code as normal. A breakpoint can be placed (or removed) by clicking in the left margin of the *Editor* window (between the code and the line number) or by using the Editor > Breakpoints menu. In both cases, when a breakpoint is active on a line a small red circle will appear in the left margin. When the code is executed the script will run until this line is reached, and then pause (with the *Editor* window active at that line). You can inspect the value of variables in the script/function by hovering over them with the mouse (a tooltip with the value appears), or by entering the variable

name at the `K>>` prompt in the *Command Window*. You can also select some code (either just a variable or a whole expression), right-click and select Evaluate Expression. The result is output to the *Command Window*. Note that a breakpoint pauses the code *before* execution of a line (so any variables on that line will not exist yet).

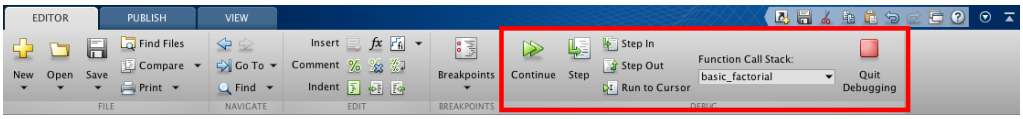

Figure 8: Debug tools

In debug mode the Editor tab on the *Editor* window contains a set of tools for controlling the execution of the script. Continue will continue running the script until the next breakpoint occurs. Step will execute the current line and then pause on the next line. Step In does the same, but if the line contains a function call then instead it will pause at the first line *inside* that function. Step Out will run the rest of the current function and will pause at the line of code after the function call. Finally, Quit Debugging will terminate the currently executing script and exit the debugger.

Literature & Resources

S. Attaway. *Matlab: A Practical Introduction to Programming and Problem Solving*. Butterworth-Heinemann, Boston, third edition, 2013. URL http://www.sciencedirect.com/science/book/9780124058767.

T. A. Davis. *MATLAB Primer*. CRC Press, Boca Raton, eighth edition, 2010.

B. R. Hunt, R. L. Lipsman, and J. M. Rosenberg. *A Guide to MATLAB for Beginners and Experienced Users*. Cambridge University Press, Cambridge, third edition, 2014.

MathWorks. MATLAB Central, 2019. URL https://www.mathworks.com/matlabcentral/. [Online].