Introduction to MATLAB

UE Numerische Mathematik 1

Wintersemester 2015

Section 1

Introduction

Introduction

MATLAB is a high-level programming language for numerical computation.

- Can be used in an *interactive* calculator-style mode
- Can also be used to write complex scripts/programs for numerical computation

In the first two weeks we shall do a *very brief* overview of MATLAB. You will potentially need to study more yourself using books/online resources. The documentation within MATLAB is also a good place to look (especially to find built-in MATLAB functions).

📄 Waltraud Huyer.
Einführung in MATLAB.
Universität Wien, 2012.
URL http://www.mat.univie.ac.at/~huyer/matlab.pdf.

📄 Peter Arbenz.
Einführung in MATLAB.
ETH Zürich, 2008.
URL http:
//people.inf.ethz.ch/arbenz/MatlabKurs/matlabintro.pdf.

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

📕 Timothy A. Davis.
MATLAB Primer.
CRC Press, Boca Raton, eighth edition, 2010.

📕 Brian R. Hunt, Ronald L. Lipsman, and Jonathan M. Rosenberg.
A Guide to MATLAB for Beginners and Experienced Users.
Cambridge University Press, Cambridge, third edition, 2014.

📕 Frieder Grupp and Florian Grupp.
MATLAB 7 für Ingenieure.
Oldenbourg Verlag, München, 2004.

📕 Frank Haußer and Yury Luchko.
Mathematische Modellierung mit MATLAB. Eine praxisorientierte Einführung.
Spektrum Akademischer Verlag, Heidelberg, 2011.
URL http://www.springerlink.com/content/h153hn/#section=786206&page=1.

🌐 MathWorks.
MATLAB Central, 2015.
URL http://de.mathworks.com/matlabcentral.
[Online].

Overview of the UI

Menu Bar Layout Workspace

Current Folder Command Window Command History
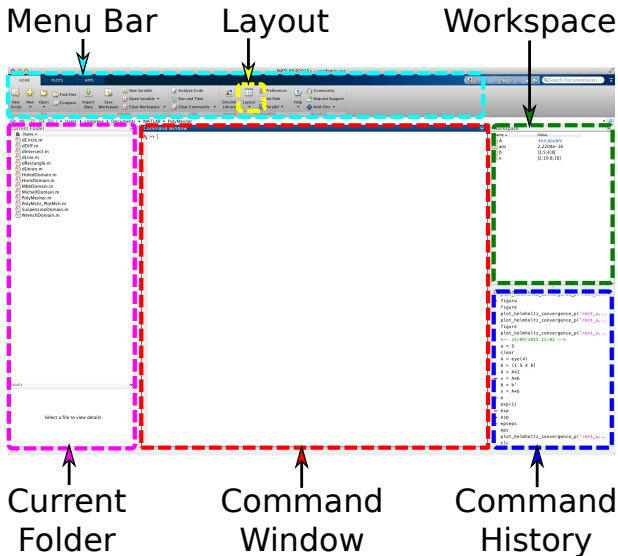
Command Window consists of a prompt (>>) at which MATLAB commands can be entered. Results are also displayed in the *Command Window*.

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 (see the *Command History* panel). Use ↑ and ↓ arrow keys on the keyboard to scroll through history.

To exit MATLAB type `exit` at the prompt.

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 is a set of built-in MATLAB directories and also the *current directory*.

Entering the command, `cd`, on its own lists the current directory.

You can also change the current directory by using

```
>> cd path
```

A special .. directory can be used to change to the *parent directory*. 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.

Documentation & Help

MATLAB has built-in documentation, which can be viewed in two different ways.

1. A graphical help window which can be launched via the UI, or by entering the command `doc`. The `doc` command can be followed by a name (MATLAB function) to display help for that function.

2. A text-based help displayed directly in the *Command Window*. Enter the `help` command to view. Again you may add a name of a function/command to display the help for that command.

Documentation & Help

help Example

```
>> 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
```

Section 2

Basic Mathematics

Scalar Arithmetic

MATLAB supports several basic scalar mathematical operators:

- \+ for addition,
- \- for subtraction,
- * for multiplication,
- / for division,
- ^ for raising to a power, and
- \ for left division (divides the second term by the first).

Using these basic commands you can use MATLAB as a calculator; i.e., entering

Basic Calculations

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

displays the result of $5^2 + 9.5 - 11 \times 2$

Scalar Arithmetic

MATLAB follows the basic mathematical rules for precedence; ^, then *
and /, and then + and -. Operators of same precedence evaluated
left-to-right. Brackets ( ) can be used to specify order of evaluation.

Evaluating $(2^2)^3$ and 2^{2^3}

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

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

Scalar Arithmetic

Multiplication symbol must be used wherever multiplication is required.

Demonstration of Requirement for Multiplication Symbol

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

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

MATLAB all numbers generated are of *double* type (by default). This means it represents floating-point numbers, but we have to allow for rounding errors in computations.

MATLAB allows numbers in an exponential (base 10) form:

- `1.5e-10` $\equiv 1.5 \times 10^{-10}$, and
- `7.95e5` $\equiv 7.95 \times 10^5$.

Number Format

By default MATLAB displays numbers in *short* form (four decimal places).

Short Format for Real Numbers

```
>> 190.2
ans =   190.2000
>> 1909.205
ans =   1.9092e+03
```

Enter `format long` to display in *long* form (15 decimal places) and `format short` for short form. See `help format` for complete list of formats.

Short Format for Real Numbers

```
>> format long
>> 19.2
ans =   19.19999999999999
>> 1909.205
ans =   1.909205000000000e+03
```

Special Constants

MATLAB has several built-in constants/special.

- `pi` returns the constant value for π,
- `eps` returns difference between 1 and next largest double number,
- `inf` or `Inf` represent ∞,
- `-inf` or `-Inf` represent $-\infty$, and
- `nan` or `NaN` represents a special *Not a Number* value.

MATLAB Constants/Special Values

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

Variables

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 is output in the *Command Window*; this can be suppressed by suffixing with a semicolon (;).

Definition and Usage of Variables

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

>> 5*x+9
ans =
      29

>> y = 9;
>> y
y =
       9
```

Variables

Several rules exist for variable names:

- Start with a letter,
- Contain letters, numbers or the underscore _ only,
- They are *case sensitive* ($A \neq a$), and
- Should not be the same as a MATLAB constant, function or keyword.

By *letters* we mean the 26 *English* letters (i.e., NO ä, ö, ü, or ß).

When working in the *Command Window* all variables are saved in the *Workspace*. Use the *Workspace* panel, or enter `who` or `whos` command, to see a list of variables.

Clear all variables from the current workspace with the `clear` command; alternatively, clear a single variable or list of variables by enter the names of the variables after the `clear` command.

Clearing Only Some Variables

```
>> clear x y
```

Complex Numbers

MATLAB also supports complex numbers. To specify an imaginary number you use `i` or `j` directly or as a suffix to a number.

Definition of Complex Number

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

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.

Calling a Function

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

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

Calling a Function with Multiple Arguments

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

Functions

Below is a non-exhaustive list of basic mathematical functions.

`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 $|x|$
`exp`	Exponential function e^x
`log`, `log10`, `log2`	Logarithmic function (e, 10 & 2)
`fix`, `floor`, `ceil`, `round`	Round: zero, down, up, nearest.
`sqrt`, `nthroot`	Square and nth root.
`angle`	Phase angle (complex number)
`conj`	Complex conjugate
`real`, `imag`	Real/imaginary parts

Enter `help elfun` for a more complete list.

Section 3

Vectors & Matrices

Defining Matrices & Vectors

The basic method to create a vector/matrix 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 `;`.

Generating Matrices/Vectors Directly

```
>> 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
```

Matrix Concatenation

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

Matrix/Vector Concatenation

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

Defining Matrices & Vectors

$start:step:end$ or $start:end$ syntax defines sequences (row vector), where

- $start$ is the first number in the sequence,
- $step$ is the difference between elements (defaults to 1), and
- end is the last number that can be contained in the sequence.

Generating Sequence Vector

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

Defining Matrices & Vectors

The *end* value is not always included in the sequence. A sequence of equally distributed values (including start and end) can be generated with the `linspace`(*start*, *end*, *no_points*) function.

Example Usage of linspace

```
>> 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
```

Defining Matrices & Vectors

Below is a list of basic matrix construction routines.

eye	Identity matrix (1 on diagonal, 0 elsewhere)
zeros	Matrix filled with 0
ones	Matrix filled with 1
rand	Uniformly distributed numbers between 0 and 1
randn	Normally distributed numbers between 0 and 1

These functions take three different formats:

- Single scalar (N) — Generates a $N \times N$ matrix
- Two scalars (N and M) — Generates a $N \times M$ matrix
- Vector of two values ($[N\ M]$) — Generates a $N \times M$ matrix

Matrix Construction Functions

```
>> ones(2,3)
ans =
     1     1     1
     1     1     1
```

Defining Matrices & Vectors

The `diag` function takes a vector and places entries on the leading diagonal of a matrix. A second optional integer allows a different diagonal to be set.

Creating a Diagonal Matrix

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

Reshaping a Matrix

The `reshape` function allows a matrix to be reshaped. This function takes a matrix as the first argument followed by a matrix size. An empty vector `[]` allows MATLAB to automatically calculate the size of that dimension.

Reshaping a Matrix

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

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

The number of elements in the reshaped matrix must be the same as in the original matrix.

An vector/matrix is accessed with element index in brackets (starting at 1). Matrices should use two indices (row and column).

A special value of `end` can be used to index the last value.

Basic Matrix/Vector Indexing

```
>> A = rand(2)
A =
    0.8147    0.6324
    0.9058    0.0975
>> A(1,2)
ans =
    0.6324
>> x = 1:5;
>> x(3)
ans =
    3
>> x(end)
ans =
    5
```

Indexing

An index can also be a vector of indices or a single : for *all* values (useful for extracting a complete row or column).

Vector Indices for Matrix/Vector Indexing

```
>> x([1 3])
ans =
     1      3
>> x(end:-1:2)
ans =
     5      4      3      2
>> A(2,:)
ans =
    0.9058    0.0975
>> A(:,2)
ans =
    0.6324
    0.0975
```

Indexing

You can change matrix entries by indexing and then assigning with =. You can delete values by assigning to them the empty matrix [].

Vector Indices for Matrix/Vector Indexing

```
>> A(2,2) = 1
A =
     0.8147     0.6324
     0.9058     1.0000
>> A(end,:3) = 0.5
A =
     0.8147     0.6324
     0.5000     0.5000
>> A(end,1:2) = [0.25 0.7]
A =
     0.8147     0.6324
     0.2500     0.7000
>> x(2:3) = []
x =
     1      4      5
```

Matrix/Vector Size

The `size` returns a vector containing the dimension of a matrix.
The `length` function returns the size of the largest dimension.

Obtaining Matrix/Vector Size

```
>> size(A)
ans =
     2       2
>> size(x)
ans =
     1       3
>> length(A)
ans =
     2
>> length(x)
ans =
     3
```

As `size` returns a two-value vector you can use the result as an argument to the matrix construction functions.

Basic Matrix Arithmetic

The basic mathematics function can be used for matrix operations.

- `+` for element-wise addition (can be applied to two matrices or a matrix and scalar),
- `–` for element-wise subtraction (can be applied to two matrices or a matrix and scalar),
- `*` for matrix-scalar multiplication or matrix-matrix multiplication,
- `/` for division of each matrix element by a scalar,
- `^` for raising a matrix to a scalar power, and
- `\` for left division of each matrix element by a scalar.

Matrix Multiplication

```
>> A = [2 0.5 1.5 4; 0 6 3 1];
>> x = [1;2;3;4];
>> A*x
ans =
    23.5000
    25.0000
```

Element-wise Arithmetic

MATLAB also supports element-wise arithmetic operators. These apply the matching scalar operation to the elements with the same index when applied to matrices of the same size.

- .* for element-wise multiplication,
- ./ for element-wise division,
- .^ for element-wise raising to a power, and
- .\ for element-wise left division.

When applied to a matrix and scalar the scalar is treated as a matrix of the same size filled with the scalar.

Element-wise Multiplication

```
>> A = [2 0.5 1.5 4; 0 6 3 1];
>> B = [3 1 4 6; 2 0 9 0];
>> A.*B
ans =
    6.0000    0.5000    6.0000   24.0000
         0         0   27.0000         0
```

Transpose

MATLAB has two matrix suffix operators (and matching functions) to take the transpose.

- ' (or `ctranspose`) transposes and takes the complex conjugate, and
- .' (or `transpose`) just transposes

Transpose

```
>> Z = [2+1i 1-8i; 0.5-0.5i 9];
>> Z'
ans =
    2.0000 - 1.0000i    0.5000 + 0.5000i
    1.0000 + 8.0000i    9.0000 + 0.0000i
>> Z.'
ans =
    2.0000 + 1.0000i    0.5000 - 0.5000i
    1.0000 - 8.0000i    9.0000 + 0.0000i
```

Solving Linear Systems

MATLAB can solve 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 $xA = B$.

Solving Linear Systems

```
>> A = [3 1 -1; 1 1 1; 0 1 -1];
>> B = [0;0;1];
>> A\B
ans =
    -0.3333
     0.6667
    -0.3333
>> B'/A
ans =
    -0.1667     0.5000    -0.3333
```

Vector Functions

Below is a non-exhaustive list of functions for vectors.

`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

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.

Matrix Functions

Below is a non-exhaustive list of functions for matrices.

`inv`	Inverse a matrix (do not use for solving linear systems)
`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

The basic mathematics functions we saw earlier can also be applied to vectors/matrices (usually element-wise).

Multiple Returns

`eig` returns multiple values. To access all returns list variable names separated by commas (surrounded by square brackets `[]`) on the left of the assignment. To ignore a return use the tilde ~ instead of a function name.

Multiple Returns

```
>> [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
```

Section 4

Strings

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

String Example

```
>> str = 'This is a test string'
str =
This is a test string
>> str(6)
ans =
i
>> str(11:14)
ans =
test
```

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

String Concatenation

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

Formatting Numbers

MATLAB has a couple of functions for converting numbers to strings.

- `num2str` converts a number to a string (4 decimal places). An optional integer argument can specify the number of decimal places, and
- `sprintf` is a more complex number formatter (check documentation).

Formatting Numbers

```
>> 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;
```

Displaying Text

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

Displaying Text

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

A `fprintf` (similar to `sprintf`) function directly outputs its result.

Section 5

Graphics

Plot Basics

The `plot` function plots x and y data as a line plot. A third argument specifies a *line specification* string: colour, marker and 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		

Plot Basics

Basic Plotting

```
>> x = 0:0.5:3;
>> plot(x,x,'-xr',x,x.^2,'-ob',x,x.^3,'-sk')
```
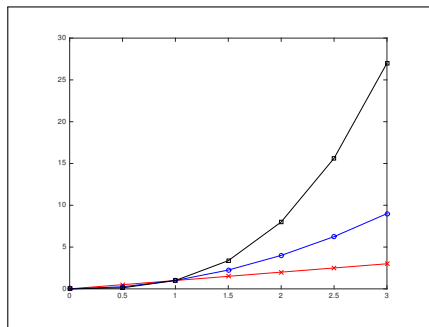

`plot` uses a linear x and y axis. `loglog`, `semilogx`, and `semilogy`, are identical but use (one or more) logarithmic axis.
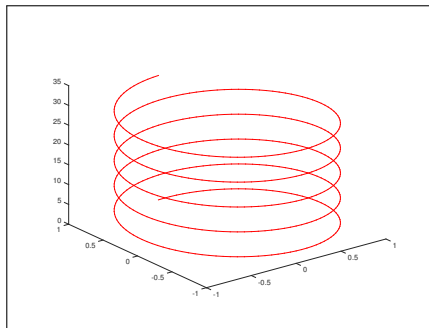
Annotation

Annotating a Plot

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

3D Plots
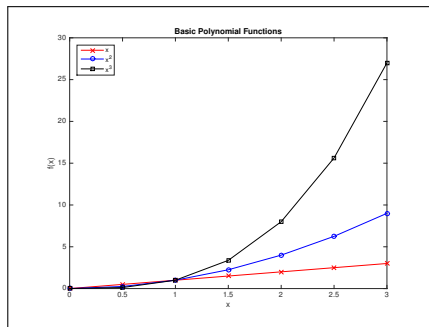
`plot3` is used for 3D line plots.

3D Line (Parametric) Plot

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


The annotations work as before (with an extra `zlabel` function).

3D Plots

`surf` is used to plot 3D data. This function takes three arguments, for the x, y and z data. z must be a $M \times N$ matrix; whereas, x and y can be a matrix or a vector.

- If x and y are matrices they must be the same size as the z matrix — matching elements from the vectors denotes a x, y and z point to plot.
- If x is a vector of length N and y vector of length M each point plotted is $(x(j), y(i), z(i,j))$.

The `meshgrid` function takes a x and y vector and creates a full x and y matrix representing the tensor points.

3D Plots

Mesh Grid

```
>> 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
```
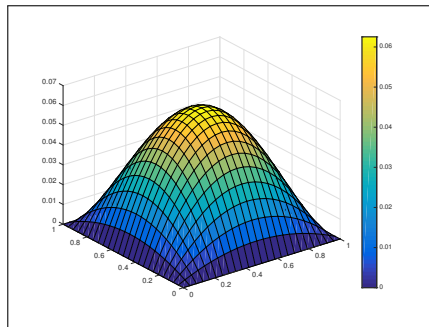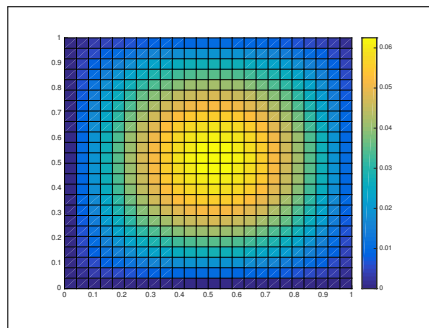
3D Plots

3D Surface Plot

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

3D Plots

The `colorbar` function displayed a colour bar on the active plot. The colour scheme used can be changed by the `colormap` function. `help graph3d` for a list available colour maps.

Rotate 3D tool-bar button (⟳) allows plot rotation by dragging within the plot area. The `view(2)` or `view(3)` function switches the current plot view to a 2D top-down view or the default 3D view, respectively.

3D Plots

Other 3D plot functions exist:

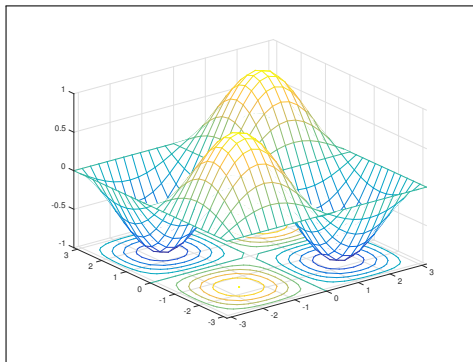|	
`mesh`	Plots mesh, rather than surface
`contour`	Plots a contour
`surfc`, `meshc`	Plots surface/mesh with contour underneath
`trisurf`, `trimesh`	Takes 3 vectors and $M \times 3$ vector of triangles (indices)

Section 6

Programming

MATLAB scripts are simple text files (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.

All variables and values in the script file are saved into the workspace.

To start editing a script file within MATLAB either:

- Create a file with | Home ⟩ New Script | or | Home ⟩ New ⟩ Script | menu bar buttons, or

- Type `edit` *filename* at the prompt (where *filename* is the name of the file). If a file with the name already exists it will be opened for editing; otherwise, a prompt will ask if you want to create a script with that name.

The *Editor* window will appear with the file to edit.

Scripts

MATLAB scripts may contain the following:

- Comments (lines not executed) in MATLAB start with a percentage sign `%`,
- Blank lines,
- MATLAB commands, and
- Block statements (`if`, `for`, `while`, `switch`, etc).

When run the script will output the result of any command not ending in a semi-colon `;`.

With scripts it is desirable to keep lines short so they can be easily read.

To split a line over multiple lines use the *line continuation ellipsis* ...
(three periods) at the end of the line to continue.

Scripts

Sample Script

```
% 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) ']']);
```

Scripts

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 the script name 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 Output

```
>> 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]
```

Functions

MATLAB allows us to define our own functions.
These are defined in a script file similar to basic scripts, but with a specific format. 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 Structire

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

end
```

Functions

A 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.

functionname   The name of the function (used when calling the function). The file name of the file containing the function must be *functionname*.m.

input_args   Comma-separated list of input argument names.

H1 Comment Line   The first comment 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.

end Keyword   The end of the function definition.

Functions

Sample Function

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

This function is called like any other MATLAB function.

Calling Sample Function

```
>> x = sample_function(2,3)
x =
      6
>> [x,y] = sample_function(2,3)
x =
      6
y =
     -1
```

Functions

While this function works for scalars it will fail with vector/matrix inputs.

Calling Sample Function with Vector Arguments

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

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

We can fix this by using the element-wise operators.

Sample Function Changed for Vector/Matrix Arguments

```
function [ z, w ] = sample_function_vec( x, y )
%SAMPLE_FUNCTION_VEC Calculates the product and difference
z = x.*y;
w = x-y;
end
```

Sub-functions

Function files can contain more than one function.
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.

Sub-functions

```
function [ z, w ] = sample_subfunction( x, y )
%SAMPLE_SUBFUNCTION Calculates the product and difference
z = multiply(x, y);
w = x-y;
end

function [z] = multiply(x,y)
%MULTIPLY Elementwise multiplication of the vectors
z = x.*y;
end
```

A special statement, `return`, can be used anywhere in a function.
This statement exits the current function and executes no more commands
from the function.

*Unlike in some other programming languages `return` does not take any
arguments (a return value), as return values are set by assignment.*

MATLAB has a built-in *logical* type, which can take a *true* or *false* value only (represented in MATLAB by 1 and 0, respectively). The following element-wise *comparison operators* return logical matrices:

$$
\begin{array}{ll}
\texttt{A < B} & \text{Checks if } A < B \\
\texttt{A > B} & \text{Checks if } A > B \\
\texttt{A <= B} & \text{Checks if } A \leq B \\
\texttt{A >= B} & \text{Checks if } A \geq B \\
\texttt{A == B} & \text{Checks if } A = B \\
\texttt{A \~{}= B} & \text{Checks if } A \neq B
\end{array}
$$

The first four work on the real part only; whereas, == and ~= works on the real and imaginary parts.

Logical Operators

Previously we mentioned that doubles suffer from rounding error.
The issue is equality comparisons do not always return true when expected.

Rounding Error in Comparisons

```
>> (19.2-19) == 0.2
ans =
      0
>> 19.2-19-0.2
ans =
    -7.216449660063518e-16
```

When comparing floating point numbers instead calculate the absolute
value of the difference and check if it less than some tolerance value.
For example, rather than evaluate `A==B`, we instead use, `abs(A-B) < 1e-8`.
The comparison operators work on numbers. The `strcmp` and `strcmpi`
perform *case sensitive* and *case insensitive*, respectively, comparisons on
two strings.

Logical Operators

The following element-wise *logical operators* and functions work on logical matrices.

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

The `all` function logically ANDs vector elements and `any` logically ORs vector elements.

Applied to a matrix the logical AND or OR is applied to each column (returning a row vector).

Short-circuit logical AND `&&` and short-circuit logical OR `||` can be applied to scalar logical arguments.

These functions are called short-circuit because they only evaluate the second argument if necessary.

Logical Indexing

A logical matrix can be used to index a matrix.

Logical Indexing

```
>> x = rand(1,4)
x =
    0.7803    0.0965    0.3897    0.1320
>> x(x < 0.5) = 0
x =
    0.7803         0         0         0
```

The `find` function returns indices of all non-zero entries in a matrix.

Find Indices of all Values < 0.02 in matrix

```
>> [i,j] = find(x < 0.02)
i =
     1
j =
     4
```

Loops are a programming structure that allows us to execute a number of statements several times.

MATLAB has both `for` and `while` loop types.

Important: 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.

for Statement

A `for` loop executes a set of statements a set number of times. Each time it executes the *index* variable takes the next value/column from an vector/matrix.

General for Loop Structure

```
for index = values
    statements
end
```

Here,

- *index* is a variable name to use as the index variable,
- *values* is the vector/matrix of values to take, and
- *statements* is the list of statements to execute each loop.

Factorial Calculation with for Loop

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

Within a `for` loop, two special statements can be used.

- `break` exits the loop completely (no more values from *values* list are evaluated), and
- `continue` stops executing the *statements* for the current value and moves to the next value from *values* (if one exists).

while Statement

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

while Loop Structure

```
while expression
    statements
end
```

- 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).

while Statement

while Loop Example

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

If the *expression* never evaluates to false then the code will be stuck in an *infinite loop*.

You can force a running script to terminate by using the Ctrl + C keyboard shortcut.

A `while` loop can contain `break` and `continue` statements.

if-else Statement

An `if` statement executes commands dependent on if a condition is true.

if-elseif-else Structure

```
if expression
    if_statements
elseif expression
    elseif_statements
else
    else_statements
end
```

The `elseif` (with following statements) and the `else` (with following statements) are both optional. Multiple `elseif` statements are also allowed.

- The `if` *expression* is evaluated. If *true*, *if_statements* are executed;
- if *false* then the *expression* for the first `elseif` statement is evaluated and *elseif_statements* executed if this is true.
- Each `elseif` *expression* is evaluated, in order, until one is true.
- If no *expression* evaluates to true then *else_statements* are executed.

if-else Statement

if Example

```matlab
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
```

A `switch` selects statements to execute based on number/string value.

switch Structure

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

The `otherwise` is optional and you can have multiple `case` statements.
`case_expression` can be single value, or multiple values (comma-separated and surrounded by braces {}).

- `switch` evaluates the *expression* and compares the result against all *case_expression*s.
- The statements of first matching *case_expression* are executed.
- If no match occurs then the *otherwise_statements* are evaluated.

switch Statement

switch Example

```
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
```

Function Handles

It is possible to store a reference to a function within a variable.
You call that function via the variable by just using the variable like a function.
To take function handle just use the function name prefixed by a `@` symbol.

Taking & Using a Function Handle

```
>> sinhandle = @sin;
>> sinhandle(pi/2)
ans =
     1
```

This allows generic functions to be written that operate on a function, without having to know what function it operates on.
The `ezplot` function takes a function handle of the function to plot.

Calling ezplot with Function Handle

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

Function Handles

We can use function handles to functions we have written as well.

Calling ezsurf with Handle to own Function

```
>> 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)
```

Previously we mentioned that functions should be written as generic as possible, we have here another demonstration of why.

Anonymous Functions

Using function handles we can also define *anonymous functions*.
This are functions that are written inline in MATLAB (usually fairly simple one-line functions).

Anonymous Function Syntax

```
@(input_args) functioncode
```

You can use this anonymous function like a normal function handle (passing to a function or assigning to a variable).
For example, we can write the `meshc` example as follows.

3D Mesh/Contour plot Using Anonymous Function

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

Section 7

Structures

Structures

A MATLAB 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:

- 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.

Generating Structure using struct Function

```
>> course = struct('Name','Numerische Mathematik 1',...
      'Group',3,'Year',2015,'Semester','WS')
course =
        Name: 'Numerische Mathematik 1'
       Group: 3
        Year: 2015
    Semester: 'WS'
```

Structures

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

Generating & Reading Structure Directly

```
>> course.Name = 'Numerische Mathematik 1';
>> course.Group = 3;
>> course.Year = 2015;
>> course.Semester = 'WS';
>> course
course =
         Name: 'Numerische Mathematik 1'
        Group: 3
         Year: 2015
     Semester: 'WS'

>> course.Name
ans =
Numerische Mathematik 1
```

Structures

Structure arrays are also possible. Structure arrays are accessed in the same way as vectors, and the array will grow to the correct size.

Accessing Structure Array

```
>> course (3) . Group = 4;
>> course (2)
ans =
          Name: []
         Group: []
          Year: []
      Semester: []

>> course (3)
ans =
          Name: []
         Group: 4
          Year: []
      Semester: []
```

Section 8

Error Handling

Understanding Error Messages

When running code occasional errors may occur. This are displayed in *red* in the *Command Window*.
Error messages usually describe the potential error fairly accurate, although the error can be caused elsewhere.
Often the error message will include a line number inside a script/function file where the error occurred. Clicking on this line number takes us to the *Editor* with the relevant file open.
See if you can spot the error in the following before running.

Function with Error

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

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

Running Function with Error

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

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

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.

If we click the line number we can then look at the code to try and find the error.

The problem is *x* is a *row vector* and it needs to be a *column vector*.

Generating Errors

You can generate error messages in your scripts (i.e., to check inputs).
To generate an error call the `error` function, passing an error message.

Generating Errors

```
function [x] = basic_factorial(n)
% basic_factorial A very basic 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
```

A similar `warning` function also exists.

Debugging

MATLAB has a debugger, which allows code to be run and inspected. The best way to enter debug mode is to place a *breakpoint* on a line of code by:

- Clicking in the left margin of the *Editor* window, or
- By using the $\boxed{\text{Editor} \rangle \text{Breakpoints}}$ menu.

When a breakpoint is active a red circle appears 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).
While debugging you can inspect values by:

- Hovering over them with the mouse.
- By entering a variable name or expression at the K>> prompt.
- Select some code, right-click and select $\boxed{\text{Evaluate Expression}}$.

Note that a breakpoint pauses the code *before* execution of a line (so any variables on that line will not exist yet).

Debugging
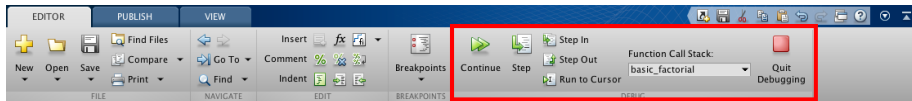
In debug mode the Editor tab on the *Editor* window contains a set of tools for controlling the execution of the script.

Continue continues running the script until the next breakpoint.

Step executes 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.

Quit Debugging will terminate the currently executing script and exit the debugger.