Intelligent Systems Lab

Michele Baldassini

Department of Information Engineering – University of Pisa

✉ michele.baldassini@unifi.it

Getting Started

What is MATLAB®?

MATLAB is a powerful software tool for:

- Performing mathematical computations and signal processing

- Analyzing and visualizing data (excellent graphics tools)

- Modeling physical systems and phenomena

- Testing engineering designs

Environment Layout and File Editor

- The **Command window** is where you type MATLAB commands following the prompt: >>

- The **Workspace window** shows all the variables you have defined in your current session. Variables can actually be manipulated within the workspace window.

- The **Command History** window displays all the MATLAB commands you have used recently – even includes some past sessions.

- The **Current Folder** window displays all the files in whatever folder you select to be current.

Running Your Code

Run the whole file from here **Run a cell from here**
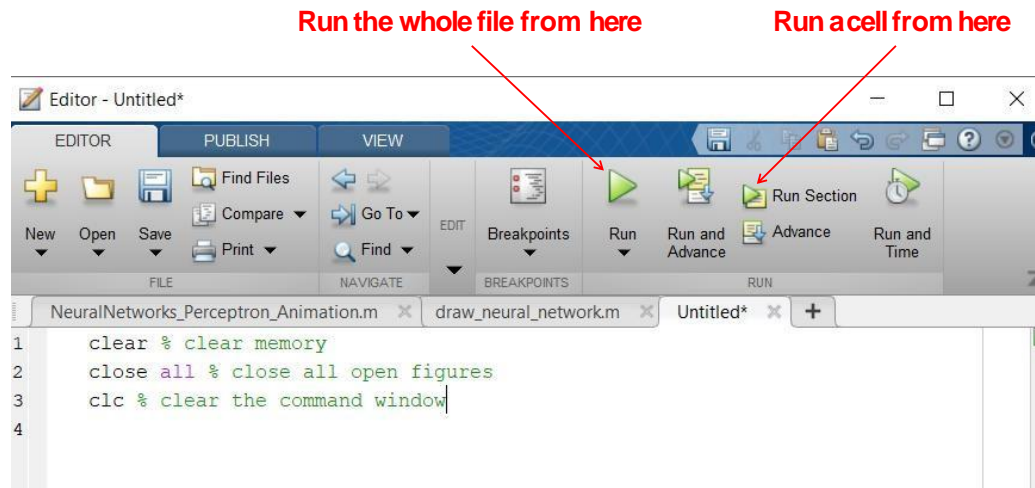
A section is delineated by a double percentage symbol followed by a space, **%%** .
By placing the cursor within the section and pressing the Run-Section icon, MATLAB will run the code from the beginning of the section to the next section or the end of the code, whichever it encounters first.

A set of MATLAB commands can be executed by one of the following ways

1. Run the file with the commands in the editor window from the run icon
   At the first run, MATLAB will ask you to save the file if you have not done this already.
   All the commands in the editor will be carried out in the order of appearance.
2. Save the file and run it by typing its name at the prompt in the Command Window.
   Make sure that the file is in the current folder or there is a path to the folder where the file is saved.
   The file names and variable names in MATLAB must not start with a number, and must not contain special symbols except for the underscore symbol.
   For example, your file can be named lab_p1.m. Then by typing lab_p1 at the » prompt, MATLAB will run the code within.

Getting Help

- Help command requires a single parameter, the name of the command you wish to get help with.

- For example, help **sin** prints the help article about the command sin directly into the Command Window starting:

        >> help sin

        SIN Sine of argument in radians.
        SIN(X) is the sine of the elements of X.

Further on you can use the various options in the Help menu.

Tips

The 'Mantra': Start your code by clearing the MATLAB memory (the workspace) using **clear**, closing previously opened figures using **close all**, and clearing the Command Window using **clc**.

```
clear        % Clear MATLAB Workspace Memory
close all    % Close all Figures and Drawings
clc          % Clear MATLAB Command Window History
```

Storing the Content of the Command Window: MATLAB command **diary <file_name.txt>** dumps the content of MATLAB Command Window in the file with the specified name, in ASCII text format. All subsequently typed commands and MATLAB answers will be stored there. To end the recording of the MATLAB window dialogue, type diary off at the Command Window prompt.

Error Messages: MATLAB displays errors in the Command Window, in red. Always read the error message. It is often a spot-on indication of what is wrong.

Good Programming Style and Design Practices

1. Strive to create readable source code through the use of blank lines, comments and spacing. It is important to put short and meaningful comments. This will help you read your code at a later date.
2. Use consistent naming conventions for variables, constants, functions and script files.
3. Use consistent indentation as provided by the MATLAB editor window.
4. Split your code into readable pieces. Use functions and separate script files where necessary.
5. Limit the creation of unnecessary variables in your code. Aim at minimum script length with maximum simplicity and clarity.
6. Where suitable, use variables instead of hard-coded values as loop limits and array sizes. This will make your code re-usable.

MATLAB as a Calculator (1/2)

Operation	Symbol	Example	Maths	Output
Addition	+	4 + 7	$4 + 7$	11
Subtraction	−	12.3 − 5	$12.3 - 5$	7.3000
Multiplication	*	0.45 * 972.503	0.45×972.503	437.6264
Division	/	5 / 98.07	$\frac{5}{98.07}$	0.0510
Power	^	4^7.1	$4^{7.1}$	1.8820e+004
Square Root	sqrt()	sqrt(15)	$\sqrt{15}$	3.8730
Logarithm*	log()	log(0.67)	$\ln(0.67)$	−0.4005
Exponent	exp()	exp(-2.1)	$\exp(-2.1) = e^{-2.1}$	0.1225
Sine**	sin()	sin(0.8)	$\sin(0.8)$	0.7174
Cosine**	cos()	cos(-2)	$\cos(-2)$	−0.4161

* natural logarithm

** the arguments for all trigonometric functions are in radians

You can type numerical expressions directly at the Command Window prompt and receive the answer after pressing Enter. The answer will be shown in the Command Window, as well as stored in a variable **ans**.
Note: **ans** is replaced by the result of each expression that is not already assigned to a variable.

MATLAB as a Calculator (2/2)

The following list of operations can be used to convert real numbers into integers:

round(a) **% Rounds a using the standard rounding rules.**
ceil(a) **% Returns the nearest integer greater than or equal to a.**
floor(a) **% Returns the nearest integer smaller than or equal to a.**

For example, **round(3.7)** = **ceil(3.7)** = 4, and **floor(3.7)** = 3. If a is an integer, then **ceil(a)** = **floor(a)** = a.

MATLAB operations may be applied to matrices as well. All operations which are done element-by-element, for example addition, subtraction and multiplication by a number, have the same syntax for both scalars and matrices. The same holds for the trigonometric functions, the logarithm and the exponent.
For the multiplication division and power, the matrix operation may be interpreted in two ways.
Hadamard product denotes an operation where the matrices are of the same size, and the entries of the resultant matrix are the pairwise products of the elements of the two matrices. MATLAB uses ***** for 'proper' matrix multiplication, and **.*** for the Hadamard product. The same holds for division and powers.
Element-wise operations are preceded by a dot, for example, **.***, **./**, **.^**

Exercises Part 1

1. Verify that the exponent (*exp()*) and natural logarithm (*log()*) are inverses of one another (cancel one another).

2. Find the root of the equation $0.5(x-2)^3 - 40\sin(x) = 0$ within the interval $[2; 4]$.

Solutions Part 1

1. Type in the command window **log(exp(10))** and then **exp(log(10))**.
   Both expressions should return the value 10.

3. Denote the left-hand side of the equation by $f(x)$. Starting with the middle of the interval ($x = 3$) find out in which half the solution lie. For example, check $x = 2$ next. As both $f(3)$ and $f(2)$ are negative, the solution must be in $[3; 4]$. Then keep dividing (and guessing, if you like) to shorten the interval of the solution until the interval length is 0.1. Return the x such that $f(x)$ and $f(x\pm0.1)$ have different signs.

Solutions Part 1

```
>> x=3;
>> 0.5*(x-2)^3 - 40*sin(x)
>> x=2;
>> 0.5*(x-2)^3 - 40*sin(x)
>> x=3.5;
>> 0.5*(x-2)^3 - 40*sin(x)
>> x=3.25;
>> 0.5*(x-2)^3 - 40*sin(x)
>> x=3.12;
>> 0.5*(x-2)^3 - 40*sin(x)
>> x=3.14;
>> 0.5*(x-2)^3 - 40*sin(x)
>> x=3.13;
>> 0.5*(x-2)^3 - 40*sin(x)
```

Since $f(3.12) < 0$ and $f(3.13) > 0$, the solution lies between the two values.
Therefore we can return either of them, say, $x = 0.12$.

MATLAB: The Matrix Laboratory

Variables and Constants (1/2)

Variables and constants (scalars) in MATLAB do not have to be declared at the beginning of the code as in other languages. Values can be assigned to them in a straightforward assignment operation, for example,

```
my_first_MATLAB_variable = -1.23456789;
m = 12;
string_example = 'My first MATLAB string';
raining_tomorrow = true;
```

MATLAB will create four variables and store the values in memory. The variables can be seen in the Workspace window of the MATLAB environment.

The semicolon at the end of the assignment operation suppresses the output in the Command Window but has no effect on the assignment itself. You can display the value of any existing variable by typing its name at the MATLAB prompt in the Command Window.

For example, typing **m** will return 12, and typing **raining_tomorrow** will return 1.

MATLAB stores *true* values as *1* but will accept any non-zero value as *'true'*; a *0* value is used for *'false'*.

Variables and Constants (2/2)

A neater way to display a string or the content of a variable in MATLAB Command Window is the command **disp**. This command takes only one argument, which evaluates to a number or a string. For example,

```
>> x = 9;
>> disp('The value of x is:');
>> disp(x);
```

MATLAB has several predefined constants that can be used in place of variables or values in expressions. The following is a small selection of some of the most useful ones:

pi	Value of π
i	Imaginary number $\sqrt{-1}$
eps	Smallest incremental number
inf	Infinity
NaN	Not a number

Names and Conventions

A valid variable name starts with a letter, followed by letters, digits, or underscores.
Note that MATLAB is case sensitive.

When choosing a variable name, it is best to avoid words which could be MATLAB commands or reserved words such as: if, end, for, try, error, image, case, plot, all, and so on. If you do this, you will (temporarily) erase the MATLAB command of the same name which you may need later in your code.

Matrices

Let us start with an example of a two-dimensional matrix (array) A with m=3 rows and n=2 columns.
The matrix can be entered into MATLAB memory (working space) as follows:

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

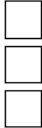The semicolon serves as 'carriage-return', separating the rows of A from one another.
The size of a matrix A can be found using **size(A)**.
For a scalar, the size command will return an array with two values: 1 (number of rows) and 1 (number of columns). For matrix A, the MATLAB answer will be 3 and 2.
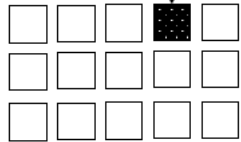
>> size(A)

Matrices

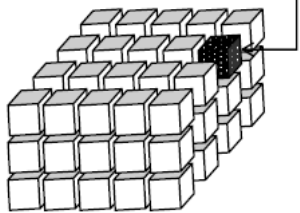Array Access from Java

-jArray[0]
-jArray[1]
-jArray[2]

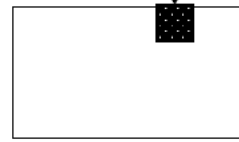Simple Array

jArray[0][3]

Array of Arrays

jArray[0][4][2]

Array of Arrays of Arrays

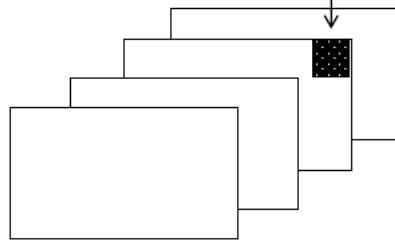Array Access from MATLAB

-jArray(1)
-jArray(2)
-jArray(3)

One-dimensional Array

jArray(1,4)

Two-dimensional Array

jArray(1,5,3)

Three-dimensional Array

Vectors are matrices where either
- m = 1, n > 1 (vector-row) or
- m > 1, n = 1 (vector-column).

Matrices may have third, fourth dimension, and so on.
For example, images can be represented as three panels for the red, green and blue colours, respectively.
Each panel is a matrix of m rows and n columns of pixels.
Hence an image is a 3-dimensional matrix of size m×n×3.

Matrices can be indexed (or subscripted in MATLAB terms) just like in other programming languages, except that in MATLAB, indices begin at 1.

Accessing Matrix Elements (1/2)

There are several ways to access individual elements of an array.
Consider matrix A from the example above. Each element can be accessed using its row and column indexes. For example, the element in row 3 column 1 holds the value 4. The line below stores this element in variable ele.

>> ele = A(3,1)

Alternatively, two-dimensional matrices can be indexed with only one index which goes from 1 to m×n. Consider a vector-column constructed by putting the consecutive columns of the matrix underneath the previous column. For example, arranging A this way will result in, the vector-column $[6; 5; 4; 3; 2; 1]^T$. Therefore, **A(5)** would return the value 2.

One of the main assets of MATLAB is that a set of elements in a matrix can be addressed simultaneously. For example:

>> A([1 4 5])

Accessing Matrix Elements (2/2)

Create a matrix L of the same size as A, containing logical values.
>> L = [true false;true true;false false]
Addressing **A** with **L**, as **A(L)** will extract only the elements of **A** where L contains a true value.
>> A(L)'

A value can be assigned to an element of a matrix by the simple assignment operator, for example:
>> A(3,1) = -9

MATLAB will also allow assignment of multiple values in one operation. For example, to replace the elements addressed through **L** by value 25, we can use:
>> A(L) = 25

Even better, you can use three different values to replace the addressed elements. For example:
>> A(L) = [55,111,222]

In addition, you can extract the desired elements in any order and with any number of copies.
For example, take two copies of element #6 followed by three copies of element #1.
>> A ([6 6 1 1 1])

Visualising a Matrix

A matrix can be visualised in MATLAB by transforming it into an image. Each element of the matrix becomes a pixel. Elements of the same value will have the same colour. Try the following code:

```
A = [1 2 3 4;5 6 7 8;9 10 11 12];    % creates matrix A
figure                                % opens a new figure window
imagesc(A)                            % transforms and shows the matrix as an image
axis equal off                        % equalises and removes the axes from the plot
```

Concatenating and Resizing Matrices (1/2)

If the dimensions agree, matrices can be concatenated using square brackets.

A = [1 2;3 4;5 6]   % matrix 3-by-2
B = [7 8;9 10]   % matrix 2-by-2
C = [A;B]   % concatenated, 5-by-2
D = [A [B;0 0]]   % concatenated, 3-by-4

A
```
1   2
3   4
5   6
```

B
```
7   8
9   10
```

C
```
1   2
3   4
5   6
7   8
9   10
```

D
```
1   2   7   8
3   4   9   10
5   6   0   0
```

A matrix can be used as a 'tile' to form a repeated pattern using the repmat command.
For example, let **A** be a 2×3 matrix. The code below uses **A** as a tile and repeats it in 3 rows and 2 columns.

>> A = [0 1 2;-2 -1 0]   % 2-by-3 matrix
>> B = repmat(A,3,2)   % 6-by-6 matrix

Concatenating and Resizing Matrices (2/2)

A matrix can be reshaped using the reshape command.
The new matrix must have exactly the same number of elements.

>> A = [1 2 3 4; 5 6 7 8; 9 10 11 12]   % 3-by-4 matrix
>> B = reshape(A,6,2)   % 6-by-2 matrix

Notice that the input matrix is first concatenated into a vector-column, taking consecutive columns and placing them underneath the last.
The output matrix is constructed column by column, taking consecutive values from the vector-column.

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

B = reshape(A,6,2)
```

number of rows
(new matrix)

number of columns
(new matrix)

concatenate
(column-by-column)

A

1	2	3	4
5	6	7	8
9	10	11	12

1
5
9
2
6
10
3
7
11
4
8
12

B

1	3
5	7
9	11
2	4
6	8
10	12
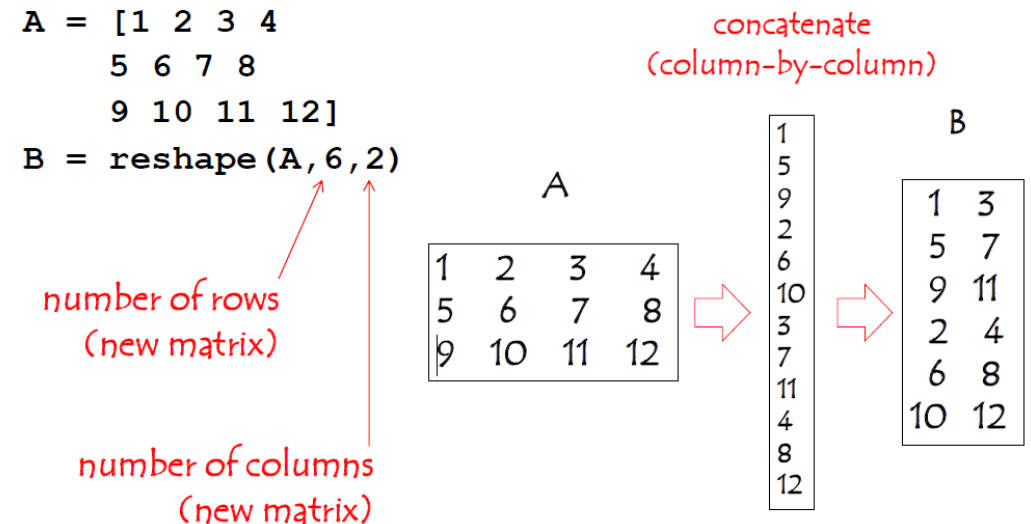
Matrix Gallery

MATLAB offers a wealth of predefined matrices. A useful subset of these is shown below:

$$
\texttt{zeros(3,2)} \qquad
\begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}
$$

$$
\texttt{ones(3)} \qquad
\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}
$$

$$
\texttt{eye(3)} \qquad
\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}
$$

$$
\texttt{rand(3,4)} \qquad
\begin{bmatrix} 0.9649 & 0.9572 & 0.1419 & 0.7922 \\ 0.1576 & 0.4854 & 0.4218 & 0.9595 \\ 0.9706 & 0.8003 & 0.9157 & 0.6557 \end{bmatrix}
$$

If a square matrix is required, one input argument will suffice, for example, **ones(3)**. This is both the number of rows and the number of columns. The identity matrix is square by definition, hence **eye(n)** only takes one input argument. Otherwise, the matrix is created with the specified number of rows and columns. These matrices can be generated in more than two dimensions.
In the matrix with random numbers, all values are drawn independently from a uniform random distribution in the interval [0,1].

The Colon Operator (1/2)

The primary use of the colon operator is to create ranges.
The basic form of a range is **start:increment:end**. The increment parameter is optional, and by default is 1.
For example, to create a vector-row x with all integers from 1 to 10, you can use:
>> x = 1:10

The three components of the colon operator do not have to be integers, nor must they be positive.
For example:
>> y = -1.7:0.81:2.452

The vectors generated by the colon operators are row-vectors. The vector contains as many elements as necessary to increase the start value by the increment value to reach the end value without exceeding it.

The Colon Operator (2/2)

Matrices can be addressed with vectors created by the colon operator.
The vector can be embedded directly as the index.
>> x = 2:4:20;
>> Y = [x;2*x]
>> Y(1,2:4)

The addressed values are in row 1, and columns 2, 3 and 4.
One particularly useful form of the colon operator is for defining the whole range within a matrix dimension.
For example,
>> Y(:,2)
will return all elements in column 2 of Y . In this case, '**:**' means 'all' (rows)

The colon operator on its own, **A(:)**, reconfigures matrix A into a column vector by placing each column below the last.

Linear spaces and mesh grid (1/2)

Instead of creating a range through **start**, **offset** and **end**, sometimes it is easier to specify the start and the end values, and the number of elements of the desired vector.
For example, to create a vector with 7 uniformly spaced elements between 0 and 1, you can use:

>> x = linspace(0,1,7)

This command ensures that $x(1) = 0$ and $x(7) = 1$.

Sometimes we need to generate all (x; y) coordinates of the points on a grid. Suppose that the grid spans the interval from 2 to 12 on x and has 4 points, and the interval from −1 and 6, and has 5 points on y.
The meshgrid command can be used for generating simultaneously the x and the y coordinates

>> [x,y] = meshgrid(linspace(2,12,4), linspace(-1,6,5))

Linear spaces and mesh grid (2/2)

Try the following example with the meshgrid and linspace commands.

```
clear all; close all; clc;
[x,y] = meshgrid(linspace(2,12,4), linspace(-1,6,5));
figure, hold on, grid on                    % open and hold a figure with a grid
surf(x,y,x.*y)                              % plot the surface of function x*y
surf(x,y,zeros(size(x)))                    % plot the plane of the zero-"ground"
stem3(x,y,x.*y,'k*-')                       % plot stems at all grid points from ground to surface
rotate3d                                    % allow for rotation of the figure with the mouse
```

Operations with matrices (1/4)

Operation	Symbol	Example	Output
Addition/Subtraction	+/−	A +/− B	Sum/Difference of the two matrices
Addition of a scalar	+/−	A − 9	Subtracts 9 from each element of A
Multiplication by a scalar	*	4*A	Multiplies every element of A by 4
Matrix multiplication	*	A*B	Matrix multiplication
Hadamard product	.*	A.*B	Element-wise multiplication of A and B

The following numerical operations are carried out on every element of the matrix:

power (.^), square root, logarithm, exponent, sine, cosine.

Operations with matrices (2/4)

Command	Return
`a'`	Transpose of a
`find(a)`	Indices of all non-zero elements in a.
`fliplr(a)`	Matrix a, flipped horizontally
`flipud(a)`	Matrix a, flipped vertically.
`inv(a)`	Inverse of a
`min(a)`	Minimum-valued element of a. †
`max(a)`	Maximum-valued element of a. †
`numel(a)`	The number of elements of a.
`repmat(a,m,n)`	A matrix where matrix a is repeated in m rows and n columns
`reshape(a,m,n)`	Matrix a reshaped into m rows and n columns.
`size(a)`	The size of a (#rows, #columns, ...)
`sort(a)`	Vector a sorted into ascending order. †
`sum(a)`	Sum of elements of a. †
`unique(a)`	The list of unique elements of a in ascending order.

† For a matrix, the operation will be carried out separately on each column. For a vector (row or column), the operation will be carried out on the vector.

Operations with matrices (3/4)

In addition to the minimum or maximum values returned by the respective commands, MATLAB returns the exact index where this value has been encountered. For example,
>> a = [3 1 9 5 2 1];
>> m = max(a)
returns 9 in **m**. If however, the command is invoked with two output arguments:
>> [m,i] = max(a)

MATLAB will return 9 in **m** and 3 in **i** because **a(3)** holds the largest value 9. MATLAB allow for replacing unnecessary arguments with a tilde (**~**). If only the place of the maximum is of interest, you can use:
[~,i] = max(a)

If a minimum or a maximum value appears more than once in the array, only the index of the first occurrence will be returned. In the example above, the minimum value 1 sits in positions 2 and 6. Only 2 will be returned as the second output argument of the min command.

Operations with matrices (4/4)

Similarly, the sort command returns as the second argument a permutation of the indices corresponding to the sorted vector. For example,
>> [s,p] = sort(a)

In this example, **s** contains the sorted **a**. **p(1)** is 2 because the first occurrence of the minimum element 1 is at position 2. The next smallest element is the 1 at position 6, hence the second entry in **p**. The third smallest element, 2, is in position 5, and so on.

Cell Arrays

A cell array is a data structure which can contain any type of data.
For example, the code below creates a cell array **C** of size 2×2 which contains strings, arrays and numerical values as its elements.

>> C = {1, 'Joey'; zeros(3), [false true; true true]}

The elements can be accessed using parentheses as with numerical arrays.
>> C(2,1)

The element is returned as a cell. If you want to access the content of the cell, use braces **{ }** instead of parentheses **( )**:
>> C{2,1}

Structures

Structures are containers for variables of different types. They are defined by simply adding element to a blank structure. For example, store the contact information of a person.

contactS.Name = 'John';

contactS.Age = 45;

The element are accessed by name

disp(contactS.Age);

One can have structure arrays, but they are tricky because each element must be a structure with the same fields

Use a structure to pass a large number of arguments to a function
- Set of parameters and prices for solving a household problem
- Our model store all fixed parameters in a structure that is passed to essentially all functions.

Structures can make code robust against changes
- Add a preference parameter to the model. Only the household code needs to be changed. Other programs use the same structure to look up model parameters.

Useful String Functions

MATLAB has string handling functions, just like any other language. As with most languages the function names (in the most part) start with *str*.

Command	What it does
`strcmp(a,b)`	Compares **a** and **b**, returning 1 if identical.
`strfind{a,b}`	Finds occurrences of **b** in **a**, returning a vector of the start positions.
`strrep(a,b,c)`	Forms a new string, replacing all instances of **b**, in **a** with **c**.
`strtok(a)`	Returns the first and the second parts of **a** split by a token.
`strtrim(a)`	Returns a copy of **a** with the leading and the trailing whitespace removed.
`isstr(a)`	Returns 1 if **a** is a string, and 0 otherwise.
`str2num(a)`	Returns the number represented by the string of digits **a**.
`input(p, 's')`	Inputs a string from the Command Window where **p** is a string prompt for the user.
`sprintf(f,d)`	Constructs a formatted string.
`disp(s)`	Output **s** in the Command Window.

Exercises Part 2

1. Using matrices, find the intersection point of the lines defined by the following equations:
$$7x - 12y + 4 = 0$$
$$12x - 45y + 26 = 0$$
   Note: Command inv(A) will return the inverse of matrix A.

2. Create a matrix A of size m×n, whose elements *a(i; j)* are calculated from the row and column indices as follows:
$$a(i, j) = (j - 4)2(i + 1) - 3 + ij$$
   The parameters m and n should be changeable. (You are not allowed to use loops here. Recall the command 'meshgrid').

3. Create vectors x and y, which, when plotted and joined, will show the pattern in figure

a

b

c
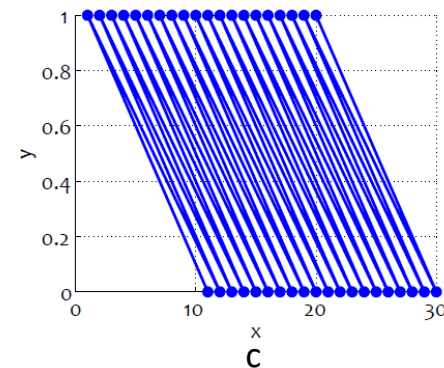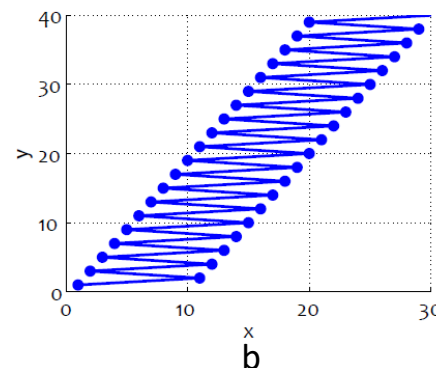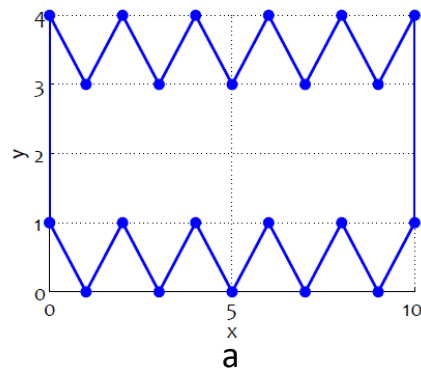
Solutions Part 2

To find the solution using matrices, we should recall that we need the matrix with the coefficients in frond of the variables (*A*) and the vector with the right-hand-side values (b):

A = [7 −12; 12 −45];
b = [-4 -26]';

The solution is $x = A^{-1}b$.

```
A = [7 -12; 12 -45];      % coefficients
b = [-4; -26];            % right-hand side vector
x = inv(A)*b;             % solution [x;y]
disp(x);
```

Solutions Part 2

2.
Suppose m = 4 and n = 3. Using 'meshgrid', we can create all row and column indices i and j:

[cols,rows] = meshgrid(1:n,1:m);

Notice that the columns are the x-coordinate, therefore they are the first output arguments of 'meshgrid', and the rows are the second output argument. With these two arrays in place, the code is:

m = 4; n = 3;
[cols,rows] = meshgrid(1:n,1:m);
A = (cols - 4).^2 .* (rows + 1).^-3 + rows.*cols;

Just to be sure, let's check with one of the values, A(2; 3)

$$A(2, 3) = (3 - 4)^2(2 + 1)^3 + 2 \times 3 = 6.0307$$

Solutions Part 2

3.
To calculate the x and y for part (a), we can use

```matlab
x = [0:10 10:-1:0];              % bottom x's followed by top x's (reverse)
y1 = [repmat([1 0],1,5) 1];      % bottom y's
y2 = y1 + 3;                     % top y's (no need to be reversed)
y = [y1 y2];                     % put the y's together
figure, plot(x,y,'b.-'), grid on
```

For part (b), y goes from 1 to 40 while x goes forth and back. We can create one of the upward lines of x's, and shift it by 10 to obtain the other. Then we need to merge them like the teeth of two cog wheels:

```matlab
x1 = 1:20;                       % left
x2 = x1 + 10;                    % right
x(1:2:40) = x1;                  % insert the left x's (odd)
x(2:2:40) = x2;                  % insert the right x's (even)
y = 1:40;
figure, plot(x,y,'b.-'), grid on
```

Part (c) is quite similar to part (b). This time y oscillates between 0 and 1. One possible solution is:

```matlab
x1 = 1:20;                       % left
x2 = x1 + 10;                    % right
x(2:2:40) = x1;                  % insert the left x's (odd)
x(1:2:40) = x2;                  % insert the right x's (even)
y = repmat([0 1],1,20);
figure, plot(x,y,'b.-'), grid on
```

Logical Expressions and Loops

Type and order of operations (1/2)

- Logical expressions may contain numerical, logical and relational operations.
  - Numerical operations involve numbers and their result is a number.
  - Relational operators compare two numbers and their result is true or false.
  - Finally, logical operations connect two logical variables. The result is again, true or false.

Relational operations

`a == b`	# which are equal to one another.
`a ~= b`	# which are not equal to one another.
`a <= b`	# where $a(i)$ is less than or equal to $b(i)$.
`a < b`	# where $a(i)$ is strictly less than $b(i)$.
`a >= b`	# where $a(i)$ is greater than or equal to $b(i)$.
`a > b`	# where $a(i)$ is strictly greater than $b(i)$.

Logical operations

`a & b`	# which are both true (non–zero).
`a && b`	Valid only for scalars. True if both are true.
`a | b`	# where either one or both of $a(i)$ and $b(i)$ are true (non–zero)
`a || b`	Valid only for scalars. True if any or both are true.
`xor(a,b)`	# where one of $a(i)$, $b(i)$ is true and the other is false.
`~a`	True if a is false. Also known as 'not' a.

```
>> a = [ 0 1 2 2 3 1 0 0 1 0];
>> b = [-2 5 0 0 4 1 0 6 3 0];
>> a&b
>> xor(a,b)
```

Type and order of operations (2/2)

Notice that logical operators join two Boolean variables while relational operators join two numerical expressions. In a logical expression, numerical operations are carried out first, then the relational operations, and finally the logical operations.

Of course, if parentheses are present, they have precedence over any operation. The operations in the innermost parentheses are carried out first, the operations within the next innermost, second, and so on. If you are not sure about the order of operations, use parentheses.

```
>> a = 2; b = 4;
```
numerical values

```
>> c =   (a - b)^2 == b   &   b >= a
```
numerical expression

```
                          4 == 4   &   4 >= 2
```
relational   relational

```
                            true   &   true
```
logical

❶

❷

❸

Therefore c contains value true

```
>> a = [4 1; 0 6]; b = [-5 1;0 4];
>> a & b              % 1
>> a ~= b             % 2
>> a - b > 3          % 3
```

Indexing arrays with logical indices

One of MATLAB's most useful features is the possibility to address an array with a logical index.
For example, suppose that we want to replace all elements of array **A** which are smaller than 0 with value, say, 22. We can create a logical index of the size of **A**, containing 1s for all elements that need to be replaced:
>> index = A < 0;

Next, we can select the relevant elements of **A** and assign the desired value:
>> A(index) = 22;

In fact, the whole assignment can be done using just one statement:
>> A(A<0) = 22;

Any logical expression that evaluates to a *true/false* matrix of the same size as that of a matrix **A**, can be used as a logical index into **A**.

Logical functions

Command	Return
`all(a)`	True if all elements of a are true/non-zero.
`any(a)`	True if any element of a is true/non-zero.
`exist(a)`	True if a exists in the MATLAB path or workspace as a file, function or a variable.
`isempty(a)`	True if a does not contain any elements.
`ismember(a,b)`	True if b can be found in a.

Conditional operations

Conditional operations act like program switches which respond to certain conditions within the program.

- As an example, consider the following tasks:
  Using the MATLAB command rand generate a random
  number between 0 and 1.
  If the number is greater than 0.5,
  display the word 'lucky', otherwise display 'unlucky'.

```matlab
if rand > 0.5
    disp('lucky!')
else
    disp('unlucky!')
end
```

- Ask the user to input an integer number between 1 and 4,
  then display the number as a word.

```matlab
n = input('Integer {1,2,3,4} = ');
switch n
    case 1 disp('one')
    case 2 disp('two')
    case 3 disp('three')
    case 4 disp('four')
end
```

```matlab
if logical_expression
    statements
elseif
    statements
elseif
    statements
...
else
    statements
end
```

```matlab
switch variable
    case value_1
    statements_1
    case value_2
    statements_2
    ...
    otherwise
    statements_n
end
```

optional

The *for* loop (1/2)

The basic syntax for a 'for' loop is:

for var = start_value : end_value
   statements;
end

var is the name of the counter variable.
var will take consecutive values starting with **start_value**
and proceeding with **start_value +1**, **start_value +2**, and so on,
until **end_value** is reached, but not exceeded.

The loop variable does not have to be an integer. Consider:
for w = -0.8:4.1

```
for i = 1:N
    <loop statements>
end
```
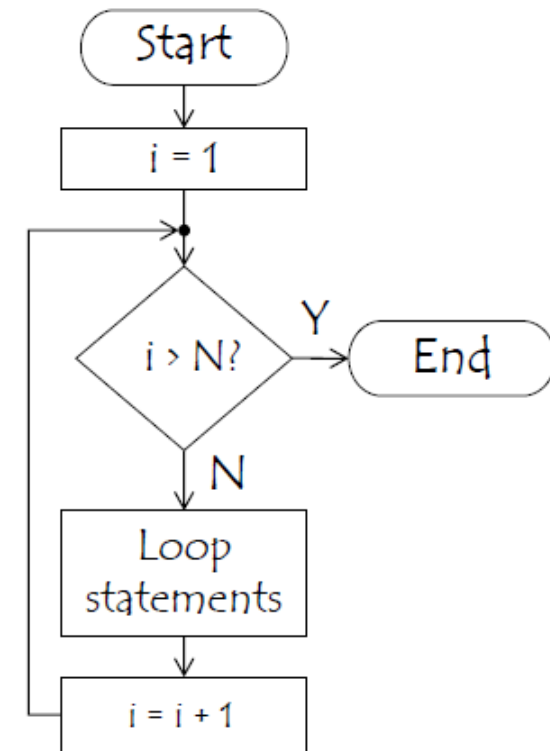
The *for* loop (2/2)

The loop variable does not have to be a number either. Consider the following example:

```
for t = 'a':'h'
    disp(t)
end
```

The default increment of the loop variable is 1. We can specify a different increment value, just as with the colon operator.

```
for w = 6.2:3.1:10.7
```

With the functionality offered by the colon operator, the for loop can run *'backwards'*.

```
for k = 12:-2:-4
```

An alternative form of the for loop, uses a matrix (or vector):

```
for var = matrix
    statements;
end
```

If matrix is a vector, **var** takes each subsequent value from the vector.

```
for v = [2 4 6 8 10 6 6 6 14]
    disp([repmat(' ',1,18-v),repmat('.',1,2*v)])
end
```
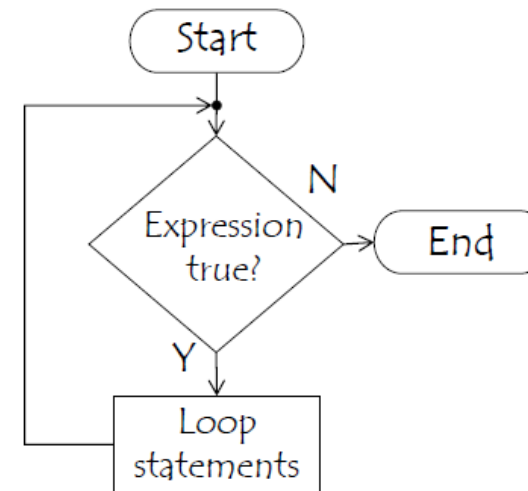
The *while* loop

The while loop uses a logical expression (condition) to determine when to exit. Whilst the expression is true, the loop continues. The statements in the loop must lead to a change in the expression value, eventually rendering it false and exiting the loop. The syntax for a while loop is as follows:

```
while expression
      <loop statements>
end
```

while expression
    statements;
end

Exercises Part 3

1. Find and display all integers between 1 and 10000 which divide by 37. Propose at least two different ways to solve this problem.

2. Write MATLAB code for the 'Guess My Number' game. First, the computer picks a random integer between 1 and 10 using the **randi** command. Next, the user is asked to enter their guess. If the guess matches the chosen number, display a congratulations message. Otherwise, display a 'better luck next time' message.

3. Consider a grid of size n×m with virtual bugs. Each bug lives in a grid cell. The grid is given to you in a form of a matrix A of size m×n, with 0s in the empty cells and 1s in the cells occupied by bugs.
    Find out the average number of neighbours per bug. Neighbours are considered to be the bugs in the 8 surrounding cells.
    To demonstrate your work, create grids of different densities using:
    A = rand(m,n) < 0.1, A = rand(m,n) < 0.5, A = rand(m,n) < 0.7

Solutions Part 3

1.
```
% first way
disp(37:37:1000)
% second way
find(~mod(1:1000,37))
% third way
x = 1:1000;
disp(x(floor(x/37)==ceil(x/37)))
% fourth way
i = 2;
z = 37;
while z < 1000
    disp(z)
    z = 37*i;
    i = i + 1;
end
```

2.
```
NumberToGuess = randi(10);
UserGuess = input('Please enter your guess -> ');
if UserGuess == NumberToGuess
    disp('Congratulations! You won!')
else
    disp('You lost! Better luck next time! The number was')
    disp(NumberToGuess)
end
```

Solutions Part 3

3.

Pad array *A* with one cell on each edge so that each cell of *A* has 8 neighbours. Then construct a double loop to go through the rows and the columns of *A*.

```matlab
m = 20; n = 30;
A = rand(m,n) < 0.1;                    % sparse
% A = rand(m,n) < 0.5;                  % medium
% A = rand(m,n) < 0.7;                  % dense
B = zeros(m+2,n+2);                     % padding
B(2:end-1,2:end-1) = A;                 % inset A
S = 0;                                  % sum of neighbours
for i = 2:m+1
  for j = 2:n+1
    if B(i,j)                           % bug
      S = S + sum(sum(B(i-1:i+1,j-1:j+1))) - 1;    % only neighbours
    end
  end
end
disp('Averge number of neighbours per bug:')
disp(S/sum(B(:)))
```

Uncomment the other versions of creating *A* to see how the average number of neighbour bugs changes.

Functions

Functions

Functions in MATLAB are stored in separate files. Unless specifically declared as global, all variables are local, which means that they are valid only within that function. Think about a function as a 'watertight' piece of code. Its communication with the outside world are the input and the output variables.

The formal syntax for a function definition is:
function [list_of_output_arguments] = function_name([list_of_input_arguments])

Neither of the lists is compulsory. If there are output arguments, they must receive values in the body of the function. An interesting property of MATLAB functions is that not all input and output argumentsneed to be assigned in the call.

For example, consider a one-dimensional array A.
```
>> m = min(A);       % will return the minimum of array A
>> [m,i] = min(A);   % will return the minimum of array A in m, and the index where the minimum is
                     % found in i
>> [~,i] = min(A);   % will return only the index of the minimum of A in i.
```

Functions

A call to **fu** is made in the MATLAB script and the output is assigned to variable **m**.
The function is stored in a separate file of the same name (**fu.m**).
It returns two outputs: the sum of the three input arguments and their product. As the function is called with one output argument only, it will return in **m** the sum 4+2+1=7.

MATLAB script

```
...
m = fu(4,2,1)

...
variable m will hold value 7 here
```

file fu.m

```
function [a,b] = fu(c,d,e)


a = c + d + e;
b = c * d * e;
```

Naming

A call to a function will look for a *file* with the function's name in the MATLAB path.
Therefore, it makes sense to set the file name and the function name to be the same.

For example, if your function is called **draw_trapeze** but the file is named **trapeze.m**, MATLAB will not accept a call to **draw_trapeze**.

It will, however, accept a call to **trapeze.m**, and will subsequently run file trapeze regardless of the name you have specified as the function declaration within the file (**draw_trapeze**).

Multiple Functions

Multiple functions can be included in the same function file. As explained above, MATLAB will refer to that collection of functions only through the file name. This structure is useful when functions in the file need to call one another.

In this example, the script calls function **fu2**.
Function **fu2** will be visible to MATLAB while function fu will not. Function **fu** is accessible only within **fu2**.
Function **fu2** returns two output arguments.
In **p**, it stores the sum of the three input arguments divided by their product, if the product is not zero.
If the product is zero, an empty value is returned (**p = []**).
The second output argument, **q**, is the sum of the sum and the product of the three inputs.
In this example, only second argument of **fu2** is requested by the MATLAB script,
therefore we store in **s** $4+2+1+4\times2\times1 = 15$.

MATLAB script

```
...
[~,s] = fu2(4,2,1)
...
```

variable **s** will hold value 15 here

file fu2.m

```
function [p,q] = fu2(x,y,z)
[v,w] = fu(x,y,z);
if w~=0
    p = v/w;
else
    p = [];
end
q = v + w;
end
%-----------------------------------

function [a,b] = fu(c,d,e)
a = c + d + e;
b = c * d * e;
end
```

function within function file fu2.m

Inline (Anonymous) Functions and Function Handles

An inline function is defined as:
fu = @(x) x.*sin(x)-exp(-x.^2);

From this point onwards, you can use **fu(arg),** where the argument can be a single number or an array.
The array operations are accounted for by using '**.***' instead of just '*****', and '**.^**' instead of '**^**'.

figure, plot(0:.3:20,fu(0:.3:20)), grid on
will open up a figure and plot the value of the function
f(x) = x sin(x) − exp(−x2) for x ∈ {0; 0:3; 0:6; 0:9; 1:2; …; 19:8}.

In the function definition, instead of a single x, we can use a list of arguments after the **@** sign
@(list_of_arguments).
fu = @(p,q) p.^q - 2*(p-q);

Recursion (1/2)

As many other programming languages, MATLAB allows recursion, which means that a function can call itself. An example is the bisection algorithm for finding a zero of a function within a given interval. Consider the function $f(x) = \sin(2x)\exp(x - 4)$

There is one zero of the function in the interval $[1; 2]$. Find the zero x^*, such that $f(x^*) \approx 0$, with precision $\varepsilon = 10^{-6}$. This means that an interval with center x^* and length ε contains the true point where $f(x^*)$ crosses the x-axis.

The algorithm goes through the following steps:
1. Input the precision, the function $f(x)$ and the bounds a and b of the interval containing the zero.
2. Calculate the middle of $[a; b]$ $m = \frac{a+b}{2}$
3. If the length of interval $[a; b]$ is smaller than ε,
   a) then return m.
   b) else, if $f(a)f(m) < 0$ (the zero is in $[a; m]$) call the function with interval $[a; m]$, else call the function with interval $[m; b]$.

Recursion (2/2)

```
function x = bisection(e,f,a,b)
   x = (a + b) / 2;
   if (b - a) > e
      if f(a)*f(x) < 0
         x = bisection(e,f,a,x);
      else
         x = bisection(e,f,x,b);
   end
end
```

The following script calls function bisection considering interval $[1; 2]$ with precision 10^{-3} :
```
e = 10^-3;                  % precision
f = @(x) sin(2*x) * exp(x-4);   % equation
xstar = bisection(e,f,1,2);
disp(xstar)
```

The MATLAB output is 1.5708.
If you substitute 1.570 and 1.571 for x, $f(x)$ should have function values with different signs.

Exercises Part 4

1. Write a MATLAB function for calculating the Euclidean distance between two two-dimensional arrays A and B given as input arguments. A is of size N×n, and B is of size M×n. The function should return a matrix D of size N×M where element d(i; j) is the Euclidean distance between row i of A and row j of B.

2. Write a MATLAB function for checking if a given point (x; y) is within the square with a bottom left corner at (p; q) and side s. The input arguments are x; y; p; q and s, and the output is either true (the point is in the square) or false (the point is not in the square).

3. Write a recursive MATLAB function to calculate the Fibonacci sequence and return the number with a specified index.
   Fibonacci numbers form a sequence starting with 0 followed by 1. Each subsequent number is the sum of the previous two. Hence the sequence starts as 0, 1, 1, 2, 3, 5, 8, 13, …

Solutions Part 4

1.
```matlab
function D = euclidean_distance_arrays(A,B)
    for i = 1:size(A,1)
        x = A(i,:);            % ith row of A
        for j = 1:size(B,1)
            y = B(j,:);        % jth row of B
            D(i,j) = sqrt(sum((x-y).^2));
        end
    end
end
```

Here is a solution without loops:
```matlab
function D = euclidean_distance_arrays2(A,B)
    N = size(A,1); M = size(B,1);
    AA = repmat(A,M,1); BB = repmat(B',1,N)';
    D = reshape(sqrt(sum((AA-BB).^2,2)),M,N);
end
```

Check with:
```matlab
P = [3 4;1 2]; Q = [-2 5;3 -1; 7 4];
disp(euclidean_distance_arrays(P,Q))
```

Solutions Part 4

2.
```
function is_in = point_in_a_square(x,y,p,q,s)
    is_in = x >= p & x <= p + s & y >= q & y <= q + s;
end
```
Here we assume that if the point is on the edge or corner, it is in the square. Check with this example:
```
point_in_a_square(0.3,0.8,0,0,1)
point_in_a_square(0.3,1.8,0,0,1)
point_in_a_square(1,0.8,0,0,1)
```


3.
```
function o = fibo_recursive(k)
    if k < 2
        o = k;
    else
        o = fibo_recursive(k-1) + fibo_recursive(k-2);
    end
end
```

Check with:
```
for i = 1:10
    disp(fibo_recursive(i))
end
```

Plotting

Plotting Commands (1/2)

The plot command is easily one of the most useful MATLAB commands. It needs at least one argument.
If there is no open figure, MATLAB will open a new one and will plot the argument (an array) versus its index.
If there are two arrays as input arguments, MATLAB will take the first array to be the x-coordinates, and the second array, the y-coordinates.

```matlab
x = -5:10; % values of the argument
y = x.^2 - 20; % values of the function
```

```matlab
figure
plot(y)
```
Plots only the function versus the index: 1,2,3...

```matlab
figure
plot(x,y)
```
Plots the function versus the argument: -5,-4,-3...

```matlab
figure
plot(x,y,'k.-')
```
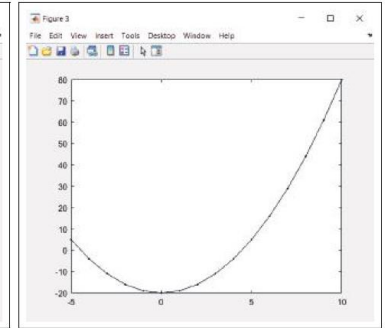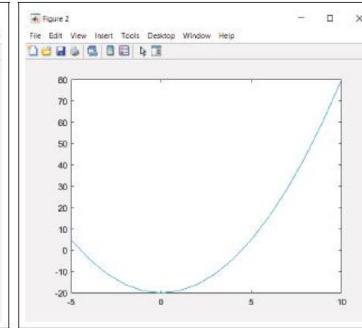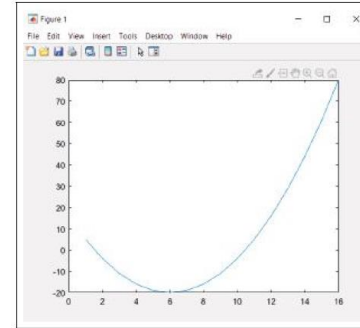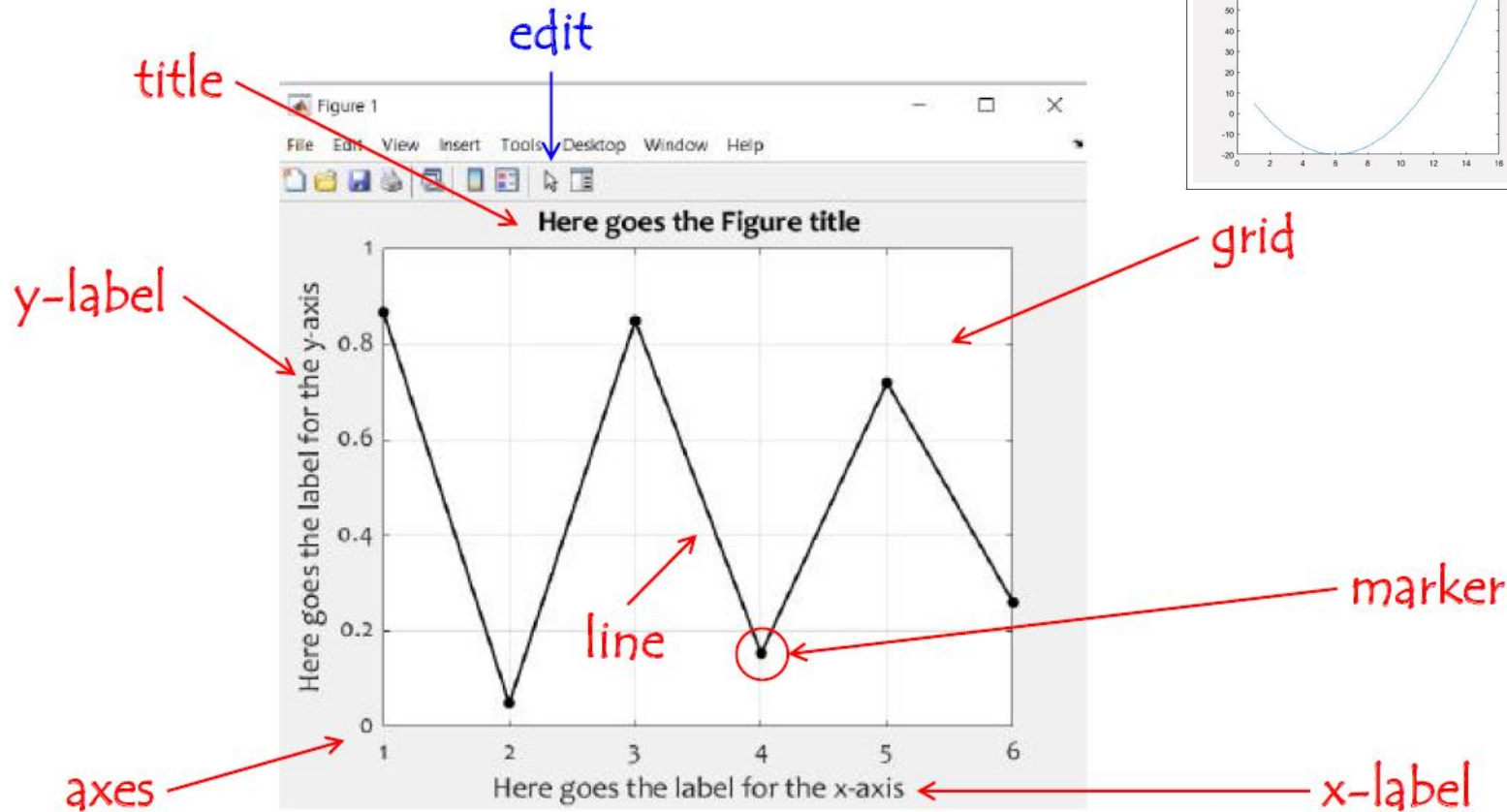Plots the function versus the argument with a black line and a black dot marker

A third string argument can specify the type of line, colour and marker of the plotted line.
If you want to plot more than one thing on the same figure, use the command hold on.
The grid lines can be toggled on and of with the command grid (use grid on).

Plotting Commands (2/2)

Fill

The fill command fills a specified region with a specified colour. The syntax is:
fill(x,y,colour)

The first argument is an array with the x-coordinates of the shape to be filled, and the second argument is an array of the same size with the y-coordinates. The third argument is either one of the pre-defined colour strings:
'k' black   'r' red   'g' green   'b' blue
'w' white   'm' magenta   'y' yellow   'c' cyan

or an array **[a; b; c]** with three numbers between 0 and 1. These three numbers make up the colour, by mixing a amount of red of possible 1), b amount of green and c amount of blue.

For example, colour 'dark salmon' is made by [0.9137, 0.5882, 0.4784].

figure, hold on
t = linspace(0,2*pi,50);   % 50 angles from 0 to 2*pi (in radians)
fill(sin(t),cos(t),[0.9137, 0.5882, 0.4784])   % circle centred at [0,0] with radius 1, filled with dark salmon colour.
grid on
axis equal

Notice two things. First, there is black outline (by default) to all 'fill' objects. Second, we used command axis equal. This makes the units on the x-axis and y-axis to become of equal size. This way the circles looks like circles and not ellipses

Exercises Part 5

1. Write a function which will draw a circle in an open figure. The input arguments are x; y; r; c; x and y are the coordinates of the centre, r is the radius, and c is a three-element vector with the colour. Demonstrate your function by using it to plot 30 circles at random positions, with random radii, and with random colours.

2. Write a MATLAB function which takes an integer k as its input argument and plots k filled squares of random colours, nested.
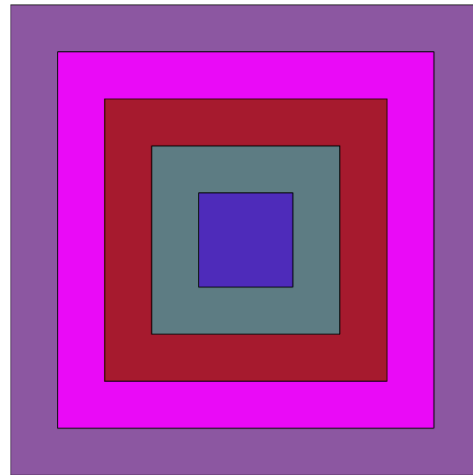
Solutions Part 5

1.
The function for plotting a circle:

```
function plot_circle(x,y,r,c)
    theta = linspace(0,2*pi,100);
    fill(x+sin(theta)*r,y+cos(theta)*r,c)
end
```

The script:

```
fig = figure;
hold on, axis equal off
for i = 1:30
    plot_circle(rand,rand,rand*0.2,rand(1,3))
end
saveas(fig, 'myFig.jpg');
close(fig);
```

2.

```
function nested_squares(k)
    hold on
    for i = 1:k
        w = k-i+1;
        fill([w,w,-w,-w],[-w,w,w,-w],rand(1,3))
    end
    axis equal off
end
```

Data and Simple Statistics

Random Number Generation

`rand`	Generates a random number with uniform distribution in the unit interval (interval [0, 1]).
`rand(n)`	Generates a square $n \times n$ matrix with random numbers in the unit interval.
`rand(m, n)`	Generates an $m \times n$ matrix with random numbers in the unit interval.
`randn`	Generates a random number from a standard normal distribution (mean 0 and standard deviation 1).
`randn(n)`	Generates a square matrix with random numbers from a standard normal distribution.
`randn(m, n)`	Generates an $m \times n$ matrix with random numbers from a standard normal distribution.
`randi(a)`	Generates a random integer from a uniform distribution between 1 and a.
`randi(a, n)`	Generates a square matrix with random integers between 1 and a.
`randi(a, m, n)`	Generates an $m \times n$ matrix with random integers between 1 and a.
`randperm(a)`	Generates a random permutation of the integers from 1 to a.
`randperm(a, k)`	Generates a random permutation of the integers from 1 to a and returns the first k elements.

Simple statistics and plots

- **Measures of Central Tendency**

mean(a)	Calculates the mean of a.
median(a)	Calculates the median of a.
mode(a)	Calculates the mode of a.

- **Measures of Variability**

 | | |
  |---|---|
  | **std(a)** | Calculates the standard deviation of a. |
  | **var(a)** | Calculates the variance of a. |
  | **range(a)** | Calculates the range of a. |

Data can be summarised and visually presented using *bar charts*, *pie charts and glyph plots*, among many. *Histograms* summarise the data by splitting the range of the variable into bins, and then counting the numbers of data points in each bin.

a = randn(1000,1); figure, hist(a)

Exercises Part 6

1. Create an imitation of a noisy signal as shown in figure. Try to reproduce the figure. Mark the minimum and the maximum of the signal with yellow square markers.

2. Write MATLAB code to do the following:
   a. Generate a random number k between 30.4 and 12.6.
   b. Generate an array A of size 20-by-20 of random integers in the interval [−40; 10].
   c. Subsequently, replace by 0 all elements of A which are smaller than k.
   d. Find the mean of all non-zero elements of A.
   e. Pick a random element from A.
   f. Visualise A using a random colour map containing exactly as many colours as there are different elements of A.
   g. Extract 4 different random rows from A and save them in a new array B.
   h. Find the proportion of non-zero elements of B.
   i. Display in the Command Window the answers of (a), (d), (e) and (h) with a proper description of each one.

3. Suppose that you are testing a slot machine. The machine has 6 types of fruit. Appearance of three of the same fruit guarantees a prize.
   a. Generate an array of 10,000 random outcomes of the three slots of the machine.
   b. Find the total number of winning combinations among the 10,000 outcomes.
   c. Assume that the entry fee for each run is 1 unit of some imaginary currency. Each winning combination is awarded a prize of 10 units except for the combination of three 1s, which is awarded a prize of 50. Assuming you are the owner of the slot machine, calculate your profit after the 10,000 runs of the game.
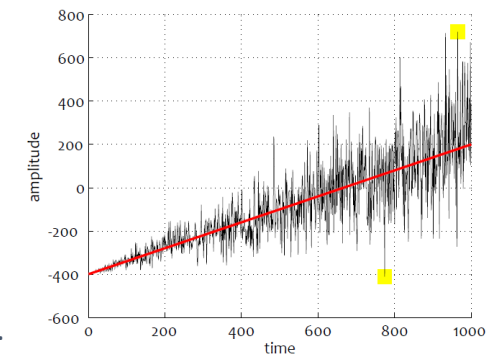
Solutions Part 6

1.

```
a = randn(1,1000).*(1:1000)*0.2;          % random signal with increasing amplitude
y_offset = linspace(-400,200,1000);        % increasing slope to add
s = a + y_offset;                          % the signal
[mins,ind_mins] = min(s);
[maxs,ind_maxs] = max(s);
figure, hold on, grid on
xlabel('time'), ylabel('amplitude')
plot([ind_mins, ind_maxs],[mins maxs],'ys','markersize',20,'MarkerFaceColor','y')
% plot the min and the max first so that the signal plot goes over
plot(s,'k-')
plot([0 1000],[-400, 200],'r-','linewidth',1.5)
axis([0 1000 -600 800])
```

Solutions Part 6

2.

a. **k = rand * (12.6 - (-30.4)) - 30.4;**
Note that the expression in the parentheses can be calculated as a single constant.
The expression was left in this form here for readability.
The random number should be multiplied by the (max - min) and then the minimum should be added.

b. **A = randi([-40, 10],20);**

c. **A(A < k) = 0;**

d. **mnzA = mean(A(A~=0));**

e. **rndA = A(randi(numel(A)));**

f. **figure, imagesc(A), axis equal off**
c = numel(unique(A));   % how many colours are needed
colormap(rand(c,3))

g. **r = randperm(size(A,1),4);   % rows # to extract**
B = A(r,:);

h. **propnzB = mean(B(:)~=0);   % B is reshaped into a vector**

i. **disp('A random number between -30.4 and 12.6:'), disp(k)**
disp('Mean of all non-zero elements of A:'), disp(mnzA)
disp('A random element of A:'), disp(rndA)
disp('Proportion of non-zero elements of B:'), disp(propnzB)

Solutions Part 6

3.

    a.   outcomes = randi(6,10000,3);

    b.   wins = sum(outcomes(:,1) == outcomes(:,2) & outcomes(:,1) == outcomes(:,3));

    c.   win_index = outcomes(:,1) == outcomes(:,2) & outcomes(:,1) == outcomes(:,3);
        profit = 10000 - sum((win_index & outcomes(:,1) ~= 1) * 10) - sum((win_index & outcomes(:,1) == 1) * 50);
        disp('Profit = '), disp(profit)

Images

Image Processing Toolbox

To check whether you have a license for the Image Processing Toolbox, type in the Command Window:

license('test', 'image_toolbox')

An answer of '1' means 'yes, you have a license'.

Binary Images

A binary image in MATLAB is a matrix containing 0s and 1s. Zeros indicate black and 1s indicate white.
Command **imshow()** will show the black-and-white image in the currently open figure.
An alternative way to show the non-zero elements of a matrix is to use the command **spy()**. This command shows the matrix on a pair of coordinate axes. The non-zero elements are plotted with blue stars.

MATLAB command | Image | Matrix representation | The spy command
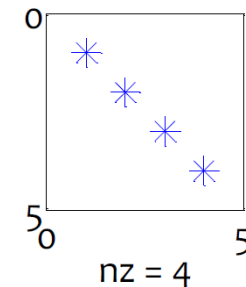
`imshow(eye(4))`

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

nz = 4

RGB Images

An RGB (red-green-blue) image is stored as three matrices (colour planes) of the same size $M{\times}N$, where M is the number of rows of pixels and N is the number of columns of pixels.

Hence, an *RGB* image A should be addressed with three indices $A(i; j; k)$, where $i \in \{1; ...M\}$ is the row, $j \in \{1; ...N\}$ is the column of the pixel, and $k \in \{1; 2; 3\}$ is the colour plane.

The colour of pixel at i,j is determined by the combination of the red intensity $A(i; j; 1)$, green intensity $A(i; j; 2)$ and blue intensity $A(i; j; 3)$. Each value is stored in *8 bits*, as *unsigned integer*, format *uint8*. Value *(0,0,0)* for $(A(i; j; 1); A(i; j; 2); A(i; j; 3))$ makes pixel $(i; j)$ black, and value *(255,255,255)*, makes it white. Equal values in the three planes will make the pixel grey, with intensity determined by that value. An example of an RGB image and its MATLAB representation are shown in figure.
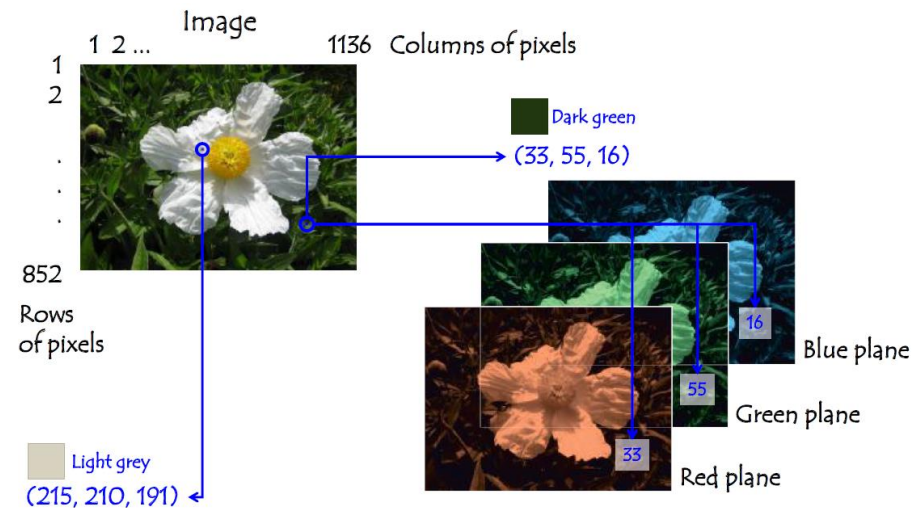
Grey Intensity Images

A grey intensity image is represented as a matrix **A** of size **M×N**. Again, **M** is the number of rows of pixels and **N** is the number of columns of pixels. Using the *uint8* format, **A(i; j)** takes integer values between *0* (black) and *255* (white), specifying the grey level intensity of the pixel in row **i** and column **j**. Each of the three planes of an RGB image is an intensity image itself.

Image

MATLAB code

```
a = repmat(0:255,700,1);
b = fliplr(a);
figure('color','w')
imshow(uint8([a b(:,1:80)]))
```
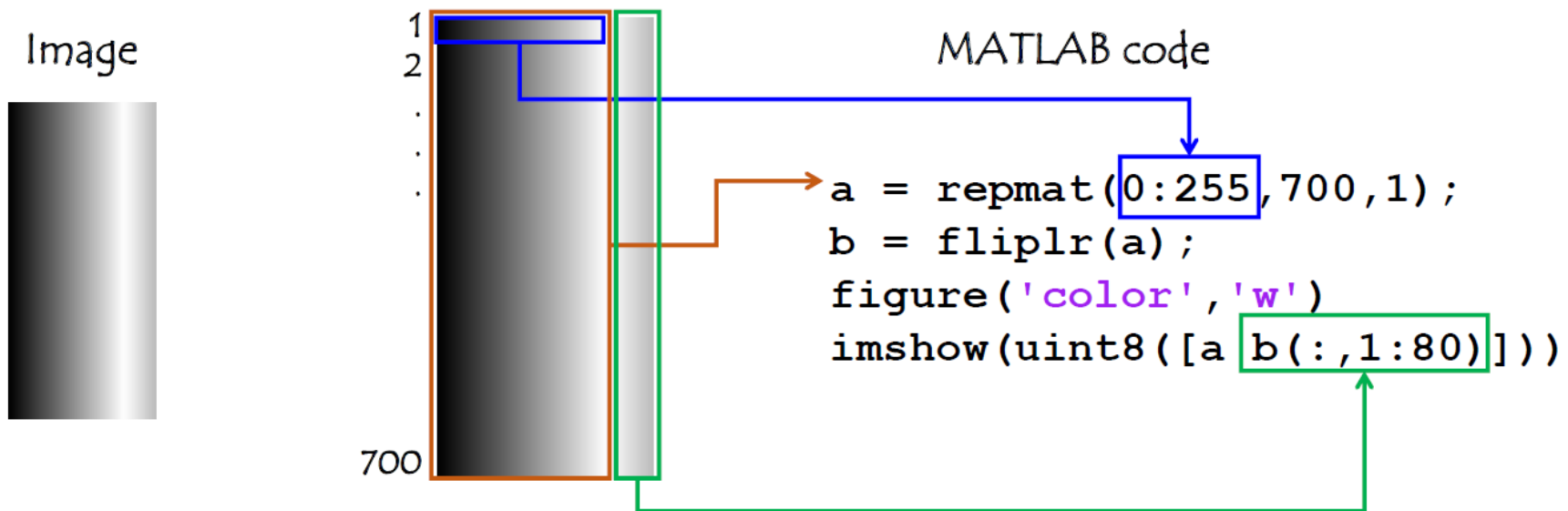
Indexed Images

An indexed image is a matrix **A** of size **M×N**, accompanied by a matrix **C** of size **k×3**, called '*the colour map*'. Each row in the colour map matrix defines a colour. The values are between *0* and *1*. White is encoded as *[1,1,1]*, and black, as *[0,0,0]*. All colours can be represented as combinations of three floating point numbers between *0* and *1*. For example, *[0.5,0,0.5]* is purple, and *[0.4,0.1,0]* is brown. The entries in **A** are taken to be row index of **C**.

Resolution: 25x46 pixels

5 colours

Colour map

```
3 3 3 1 4 4 4 0 4 4 2 2 4
3 3 2 4 0 0 0 0 0 0 0 2 0
1 2 4 0 0 2 2 2 2 2 0 0 2
2 4 2 0 0 2 1 1 1 1 0 0 2
3 3 3 0 0 2 2 1 1 2 2 0 0
3 3 3 1 0 0 1 2 2 2 0 0 2
3 3 3 3 0 0 0 1 2 0 0 2 3
3 3 3 3 3 0 0 0 0 0 1 3 3
3 3 3 3 3 3 3 1 1 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3
```

0	0.07	0.08	0.18
1	0.70	0.71	0.86
2	0.36	0.38	0.64
3	0.99	0.99	0.99
4	0.06	0.07	0.55
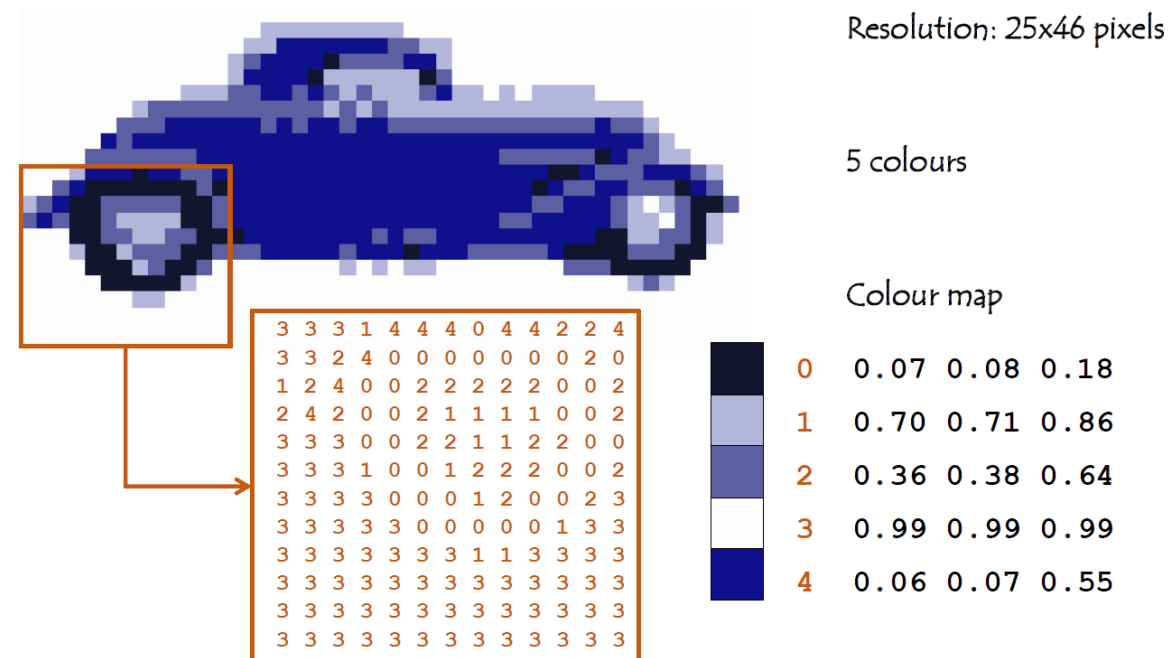
Useful Functions

To extract the three planes from an RGB image A, use:

R = A(:,:,1);   % red plane
G = A(:,:,2);   % green plane
B = A(:,:,3);   % blue plane

To (re-)assemble an image from
three planes, R, G, and B, use:

A = cat(3,R,G,B);

This command will concatenate the three planes on the third dimension.

Command	What does it do?
`imread(i)`	Loads an image into a matrix.
`imshow(A)`	Displays an image matrix in a figure.
`imagesc(A)`	Scales a matrix into an image and displays the image.
`colormap(m)`	Sets the active colormap.
`rgb2gray(A)`	Converts an RGB image into a grayscale image.

Sometimes it is necessary to store the coordinates of the pixels in an image.

[x,y] = meshgrid(1:5,1:3);

Notice that the y coordinate starts from top and increases with the row index. This is the ij-coordinate system available in MATLAB. To plot in a figure, with this system, set the axes by axis ij.
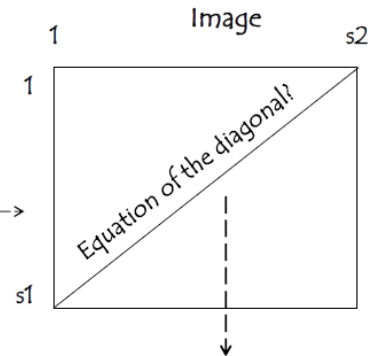
Image Manipulation: Example

Load an RGB image and display it so that the top diagonal half is grey, and the bottom part is unchanged.

```
A = imread('myImg.jpg');        % load the RGB image in matrix A
B = rgb2gray(A);                % convert to grey
s = size(B);
```


Hmm, how do I solve this problem?...

1. Upload the RGB image in variable A.
2. Convert to grey and store in B.
3. Find the diagonal that should split the two halves.
4. Run a double loop and check the pixel coordinates with the left-hand side (LHS) of the equation. If negative, assign the grey value to the three colour planes.
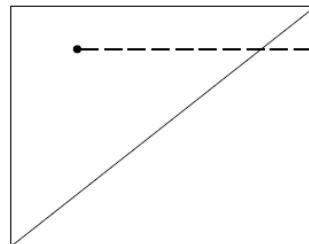
Image

For pixel (i,j)

Substitute x=j, y=i.
If LHS < 0, set
A(i,j,1:3) = B(i,j)

Two pixel coordinates (1,s1) and (s2,1)
Equation:
$(x-1)/(s2-1) = (y-s1)/(1-s1)$
$(x-1)/(s2-1) - (y-s1)/(1-s1) = 0$

Above diagonal: (–) side
Checked with (1,1)

Image Manipulation: Example

```matlab
for i = 1:s(1)
    for j = 1:s(2)
        if (j-1)/(s(2)-1) - (i-s(1))/(1-s(1)) < 0;        % top half
            A(i,j,:) = B(i,j);
        end
    end
end
figure,imshow(A)
```

While this approach works; it is slow, and not in the spirit of the language. Instead of the double loop, we can create a mask which will have values *true* for the top diagonal half. Then we will replace the top diagonal halves of the RGB planes of A with the corresponding values in B, and finally re-assemble A. The code for this version is shown below.

```matlab
[x,y] = meshgrid(1:s(2),1:s(1));                % x-y coordinates for all pixels
mask = (x-1)/(s(2)-1) - (y-s(1))/(1-s(1)) < 0;  % top half
r = A(:,:,1); r(mask) = B(mask);                % red plane
g = A(:,:,2); g(mask) = B(mask);                % green plane
b = A(:,:,3); b(mask) = B(mask);                % blue plane
figure,imshow(cat(3,r,g,b))                      % open a figure and show concatenated image
```

Exercises Part 7

1. Take a JPEG image and tint the four quadrants with transparent overlays

2. Load a JPEG image and draw a grid with 10 rows and 10 columns of cells on it. The grid lines should be embedded in the image, and not merely plotted on the same axes. The width of the lines should be chosen in such a way that the lines are visible. Also, make your code re-usable so that it can work on any image you upload. (This means that there should be no hard coded constants in your function/script).

2. Construct the function shuffle_image that will take an RGB image and two integers, M and N. The function should split the image into M rows and N columns of 'tiles'. It should return image of the size of the original input but with shuffled tiles. An example of the original image and the shuffled image, for M = 4 and N = 5.
   Note: If needed, make the image sizes multiples of M and N, respectively, by losing a small number of bottom rows and right-hand side columns of pixels.

Solutions Part 7

1.

```matlab
A = imread('myImg.jpg');
s = size(A);
midR = round(s(1)/2); midC = round(s(2)/2);
A(1:midR,1:midC,1) = 255;              % red top left
A(1:midR,midC+1:end,3) = 255;          % blue top right
A(midR+1:end,1:midC,2) = 255;          % green bottom left
A(midR+1:end,midC+1:end,[1 3]) = 255;  % purple bottom right
figure, imshow(A)
```

2.

```matlab
A = imread('myImg.jpg');
figure, imshow(A)
s = size(A);
lw = ceil(s(1)/200);
tenrows = round(linspace(1,s(1)-lw,11));
tencolumns = round(linspace(1,s(2)-lw,11));
B = A;
for i = 1:lw
    B(tenrows+i-1,:,:) = 0;
    B(tenrows+i-1,:,2) = 255;
    B(:,tencolumns+i-1,:) = 0;
    B(:,tencolumns+i-1,2) = 255;
end
figure, imshow(B)
```

Solutions Part 7

3.
The script:

```matlab
function Im = shuffle_image(A,M,N)
    if ismatrix(A)                              % is a matrix
        A = cat(3,A,A,A);                       % make a grey image into rgb
    end
    Rows = floor(size(A,1)/M);
    Columns = floor(size(A,2)/N);
    Im = uint8(zeros(size(A)));
    % Shuffle index
    RP = reshape(randperm(M*N),M,N);
    k = 1;
    for i = 1:M
        for j = 1:N
            T = A((i-1)*Rows + 1 : i*Rows,(j-1)*Columns + 1 : j*Columns,:);          % take current block
            [new_r,new_c] = find(RP == k);
            % position in the new row/column
            Im((new_r-1)*Rows + 1 : new_r*Rows,(new_c-1)*Columns + 1 : new_c*Columns,:) = T;
            k = k + 1;
        end
    end
end
```

The script:

```matlab
A = imread('myImg.jpg');
B = shuffle_image(A,4,5);
figure,imshow(A)
figure,imshow(B)
```
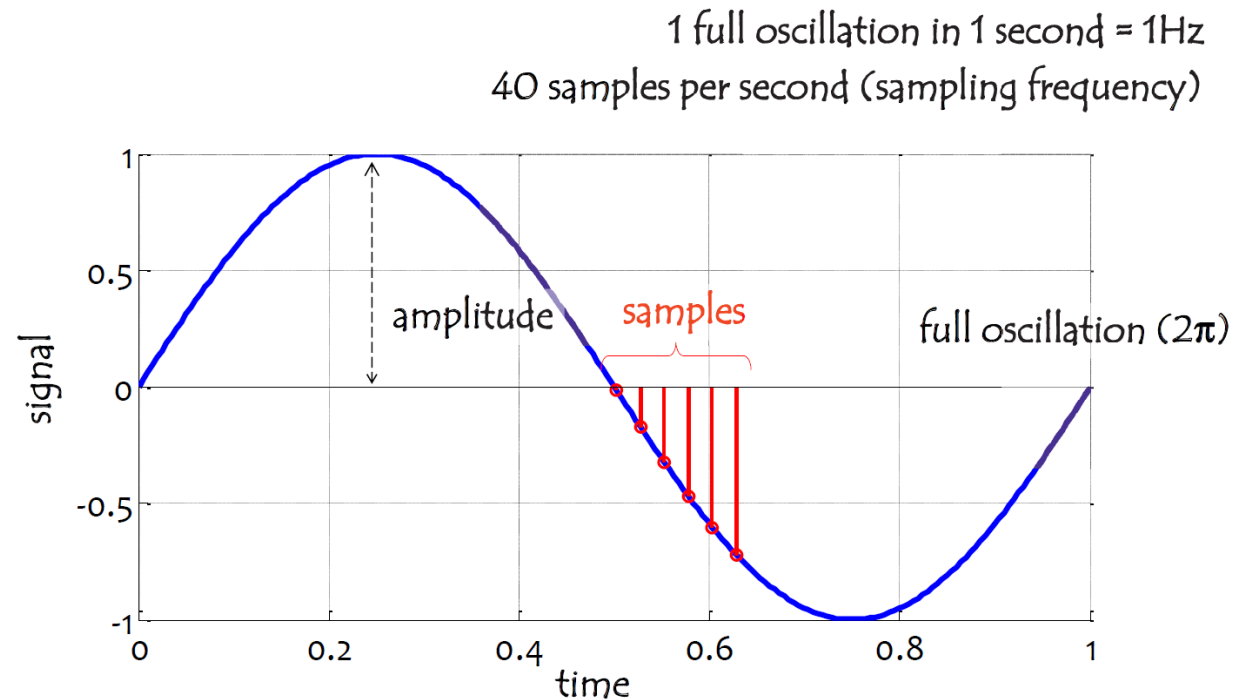
Sounds

Sounds as Data

Sound is made up by waves which are often simplified to a sine function. The sine way is characterized by amplitude, frequency and phase. The phase is important when a sound contains more than one waves and they are offset by their phases. The amplitude determines how loud the sound is, and the frequency determines the pitch.

Usually sounds are much more complicated than a single sine wave, including many sine-like waves together.
A sound can be reproduced if we find the sine waves it is made up from.

MATLAB command **audioread** reads sound data and returns sampled data *y*, and a sample rate for that data *Fs*.

1 full oscillation in 1 second = 1Hz
40 samples per second (sampling frequency)

amplitude   samples   full oscillation (2π)

Example 1

Sound can be created as a sine function and played in MATLAB using the sound command.

- Creates and plays sound with a sine function with frequency $F = 440 Hz$ for $T = 2\ seconds$. Sounds sampled with one the standard frequencies, $fs = 8000\ samples\ per\ seconds$.

```
fs = 8000;          % the sampling frequency
T = 2;              % length of the note in seconds
t = 0:1/fs:T;
F = 440;            % frequency of sine
y = sin(F*2*pi*t);  % the signal
sound(y,fs)
```

Example 2

- Consider another example, where a sounds is played for $3\ seconds$ while fading away. In this case the amplitude should gradually decrease to zero while the frequency will stay unaltered.

```
fs = 8000;              % the sampling frequency
T = 3;                  % length of the note in seconds
t = 0:1/fs:T;
F = 523.25;             % frequency of sine
A = (T - t)/T;
y = A .* sin(F*2*pi*t);  % the signal
sound(y,fs)
```

Note the element-wise multiplication where each value of the sine signal is multiplied by the respective amplitude value.

Exercises Part 8

1. Write a script to play an ascending musical scale. Each note should be played for *0.8 seconds* and should fade linearly. Include overtones.
2. Add a second melody playing the scale backwards (descending), four times quieter than the leading melody. The harmony should be played together.
3. Modify the melody you created in *(2)* so that it 'speeds up'. For example, if the first note lasts *1 second*, the last one should last *0.1 seconds*.
4. Plot the first *30 milliseconds* of the sound signal in *(2)*. Label the axes and add a title.
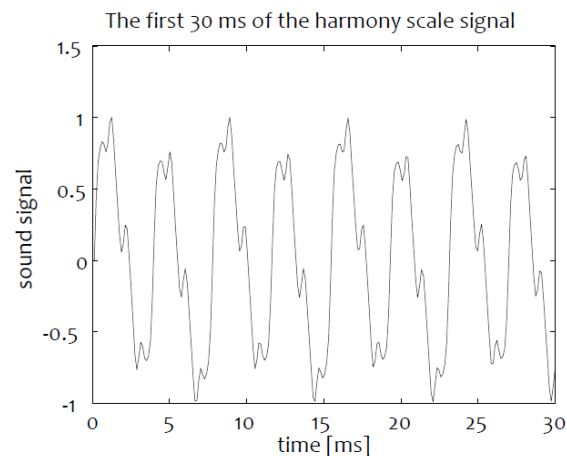
Solutions Part 8

A piece of code needed for all sub-problems

```matlab
noteFrequency = [261.63 293.66 329.63 349.23 392.00 440.00 493.88 523.25];
fs = 8000;              % sampling frequency
```

1.
```matlab
    y = [];
    T = 1;              % time in seconds
    t = 0:1/fs:T;
    A = (T - t) / T; % fading amplitude
    for i = 1:8
       no = noteFrequency(i);
       s = A .* (sin(2*pi*t*no) + 0.2*(sin(pi*t*no) + sin(4*pi*t*no)));
       y = [y, 0 s];
    end
    sound(y,fs)
```

2.
```matlab
    y = [];
    T = 1;
    t = 0:1/fs:T;
    A = (T - t) / T;
    for i = 1:8
       no = noteFrequency(i);
       noback = noteFrequency(9-i);
       s1 = A .* (sin(2*pi*t*no) + 0.2*(sin(pi*t*no) + sin(4*pi*t*no)));
       s2 = A .* (sin(4*pi*t*noback) + 0.2*(sin(2*pi*t*noback) + sin(8*pi*t*noback)));
       y = [y, 0 0.8 * s1 + 0.2 * s2];
    end
    sound(y,fs)
    audiowrite('scales_harmony.wav', y, fs)
    yb = y;          % save the signal for the plot in 2(d)
```

Solutions Part 8

3.

```
y = [];
T = linspace(1,0.1,8);
fs = 8000;
for i = 1:8
  t = 0:1/fs:T(i);
  no = noteFrequency(i);
  noback = noteFrequency(9-i);
  A = (T(i) - t) / T(i);        % amplitude is specific for duration T(i)
  no = noteFrequency(i);
  noback = noteFrequency(9-i);
  s1 = A .* (sin(2*pi*t*no) + 0.2*(sin(pi*t*no) + sin(4*pi*t*no)));
  s2 = A .* (sin(4*pi*t*noback) + 0.2*(sin(2*pi*t*noback) + sin(8*pi*t*noback)));
  y = [y, 0 0.8 * s1 + 0.2 * s2];
end
sound(y,fs)
```

4.

```
% fs per second, fs/1000 per milisecond, round(30*fs/1000) per 30 ms
N = round(30*fs/1000);    % number of samples corresponding to 30 ms
X = linspace(0,30,N);       % prepare x-axis
figure, plot(X,yb(1:N),'k-')
set(gca,'FontName','Candara','FontSize',12)
title('The first 30 ms of the harmony scale signal')
xlabel('time [ms]')
ylabel('sound signal')
```