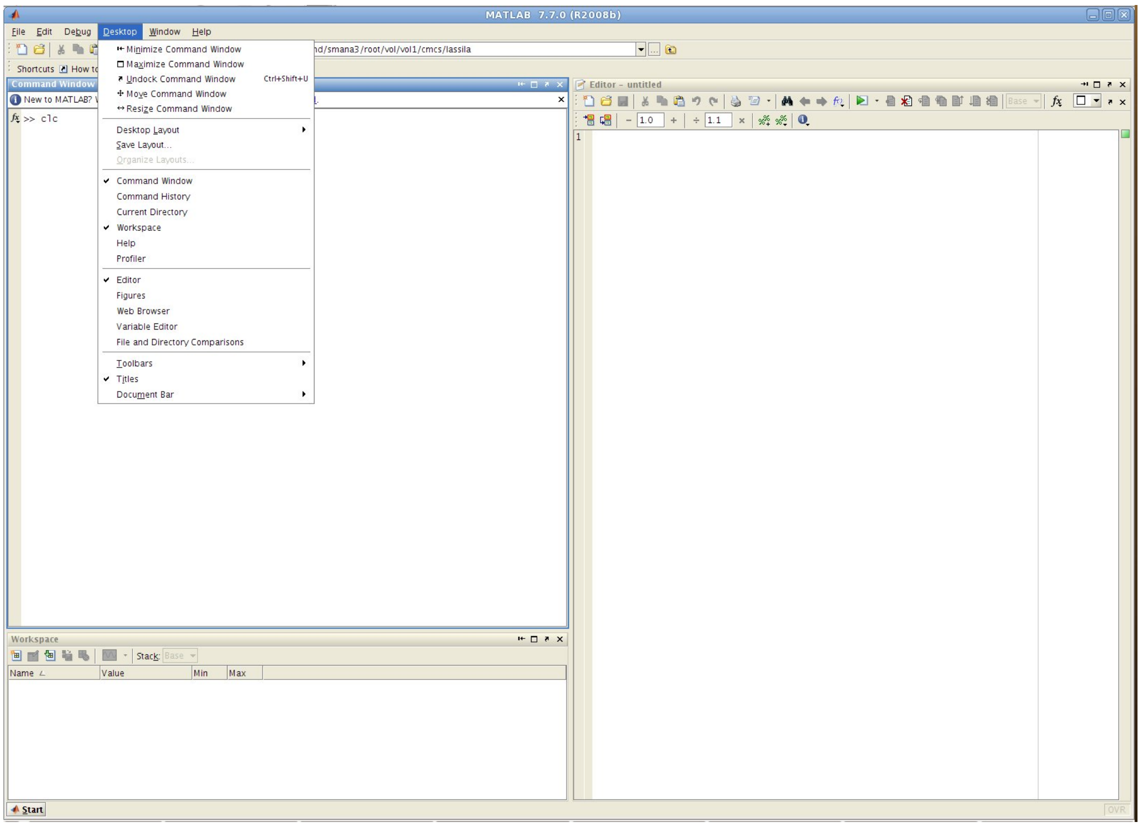
1 Setting up MATLAB



```
>> clc
```

To start MATLAB click on Start/Programmes/MATLAB. MATLAB opens as a Graphical User Interface (GUI) – complete desktop environment – for interactive usage. When you launch MATLAB for the first time, the interface tends to be cluttered with many panels open by default. To reduce the clutter, use the menu **Desktop** to disable all the panels except the three most important ones: **Command Window**, **Workspace**, and **Editor**. Note that you can drag-and-drop the panels into different positions and resize them as you wish. Try to replicate the configuration displayed in the screenshot above.

The **Command Window** is your interactive way to give commands for the MATLAB interpreter to run. A prompt '>>' appears in the Command Window indicating we are ready to type commands. Each command is completed by pressing `Enter`. Try it now:

```
>> 1+1

ans =

     2
```

The MATLAB interpreter takes your command, evaluates it, and returns the value in a variable called `ans`. In this case, the value of `ans` equals 2. MATLAB directs output into the command window by default. To suppress output, add a semicolon '`;`' to the end of the command. In this case MATLAB does not print the value of the computation, but it is still stored in a variable called `ans`.

The **Workspace** stores in memory all the variables you have defined during your MATLAB session. Once you close MATLAB, your workspace will be lost. To save your workspace before you quit, you can use the `save` command. To load your workspace, use the `load` command. To learn more about these commands, use the `help save` and `help load` commands. The currently defined variables in your Workspace are listed in that panel for easy reference.

Instead of writing every command directly to the Command Window, it is more convenient to use the **Editor** to write script files (called `.m` files) that are stored in a subdirectory of your home folder and can be reused later. Simply write all the commands you want to run in one batch into the editor, then save it as a file without forgetting to add the `.m` suffix (since otherwise MATLAB cannot find the file). For example, if you have a file called `myscript.m` in your working folder[1], then giving the command

```
>> myscript
```

in the Command Window will execute all commands contained in that file.

To get more help on any of the MATLAB built-in commands, use the `help` command. You can also browse the graphical **Help** system by giving the command

```
>> doc
```

and discover everything that MATLAB can do (and that's a lot).

2 MATLAB data types and working with matrices

MATLAB has few basic data type: **booleans, integers, floating point numbers** (real or complex), and **strings**. Each variable must have a unique name that:

1. must begin with a letter and cannot have spaces in it (variable names cannot begin with a number);
2. are case-sensitive (`MYVALUE` and `MyValue` are different variables);
3. is not a reserved keyword or the name of a built-in-function.

Variables are declared automatically with the correct data type as you initialize them. To initialize a variable, use the assignment operator `=` like this:

```
>> x = true;
>> y = 42;
>> z = 7.894e5
>> v = 1 + 2*1i;
>> w = 'hello world';
```

Note how after you initialize each variable it appears in your Workspace panel with the current value indicated. This is a good way to check that your variables contain the values that you want them to.

Once you have defined some variables, you can perform basic operations on them. MATLAB understands all the basic algebraic operations:

```
>> y + 2 * (y - 1)

ans =

   124
```

It can also compute most special functions without problems:

```
>> sin(z) + cos(z) + exp(-z)

ans =

    0.3265
```

[1]The working folder is displayed on top of your MATLAB desktop. If your MATLAB interpreter cannot find your file, you may have to manually navigate to the correct directory. Note that by default MATLAB only looks for files in the current working directory.

If you want to see more decimals in the floating point presentation, use the `format` command:

```
>> format long;
>> ans

ans =

   0.326483382481923
```

Other useful formatting options include `format long e`, `format long eng`, and `format rat`. Then go back with

```
>> format short
```

Note that `1i` is a preset symbol denoting the complex unit, $i^2 = -1$. You can equivalently use `i` or `j`. To take just the real or imaginary part of a complex number do this:

```
>> real(v)

ans =

     1

>> imag(v)

ans =

     2
```

For floating point computations MATLAB defines the useful function `eps(x)`, defining the **machine epsilon** at `x`. It measures the positive distance from x to the next larger in magnitude floating point number of the same precision as x. In standard architectures and double precision, `eps(1)` $= 2^{-52} = 2.2204\text{e-}16$. Note that `(1 + eps(1)) > 1`, but that `(1 + eps(1) / 2) == 1`, since floating point arithmetic is unable to represent differences smaller than `eps`.

3 Basic operations on matrices

MATLAB's basic premise is that every variable can be considered as a rectangular $n\times m$ array of numbers i.e. a matrix – scalars are just 1×1 matrices, and vectors are just $n \times 1$ or $1 \times m$ matrices. To create a matrix of specific elements, use the notation

```
>> A = [1 2; 3 4]

A =

     1     2
     3     4
```

To create a row matrix, use

```
>> b = [5 6];
```

Note that when working with matrices and vectors, the MATLAB default algebraic operations `+`, `-`, `*`, `^` are those of matrix algebra. The usual matrix operations are as follows:

`>> C = A + B`	$C_{ij} = A_{ij} + B_{ij}$
`>> C = A * B`	$C_{ij} = \sum_k A_{ik}B_{kj}$.
`>> C = A / B`	$C = AB^{-1}$
`>> C = A^3`	$C = A \cdot A \cdot A$

Remember that matrix operations are well defined only if the sizes of the matrices are mathematically coherent. To perform elementwise operations on matrices, use instead the dotted operators `.*`, `./`, `.^`:

`>> C = A .* B`	$C_{ij} = A_{ij}B_{ij}$
`>> C = A ./ B`	$C_{ij} = A_{ij}/B_{ij}.$
`>> C = A.^3`	$C_{ij} = A_{ij}^3$

There are a couple of useful functions that return commonly needed matrices: Let `m`, `n` be two integers, `v` a column vector, and `a`, `step`, `b` scalars. Then:

```
>> A = eye(n)             is the n x n identity matrix

>> A = diag(v)            is a diagonal matrix with v on the diagonal

>> A = zeros(n,m)         is a full n x m matrix filles by zeros

>> A = ones(n,m)          is a full n x m matrix filles by ones

>> x = [a:step:b]         is a row vector with equidistributed points with values
                          [a, a + step, a + 2step, ...] with the last element <= b.
```

Note that the function `diag` has also another purpose. If the argument is a square matrix, `diag(X)` return a row vector containing the diagonal of the matrix X. A sometimes useful trick is to write `diag(diag(X))`. This returns a matrix of the same size as X, but with all the nondiagonal elements set to zero. To similarly extract either the upper-diagonal or lower-diagonal parts of a square matrix, use `triu` or `tril`.

Any matrix `A` of size $n \times m$ in MATLAB can be accessed by the index-notation `A(i,j)` that returns the (i, j):th element of the matrix, where $1 \leq i \leq n$ and $1 \leq j \leq m$. This can also be used to assign values to the elements of the matrix one-by-one. However, this is usually not very efficient and it is best to avoid looping over matrix/vector elements whenever possible. Use instead the powerful MATLAB indexing operator `:` as follows:

```
A(:,1)           returns the first column vector of A
A(2,:)           returns the second row vector of A
A(:,2:end)       returns all the column vectors of A except the first one
A(end-1,end-1)   returns the element (n-1,m-1)
A(100)           returns the 100th element of A
```

By using the indexing operators you can often express your computations in vectorial form without needing to loop over individual elements. This is typically orders of magnitude faster in MATLAB. To facilitate this, sometimes it is useful to take an $n \times m$ matrix `A` and reshape it into a vector of size $nm \times 1$ given by `A(:)` where we have concatenated all the column vectors one-after-another into one long vector. Use the `reshape` command to:

```
>> A = [1 2; 3 4];
>> A(:)'

       ans =

         1 2 3 4

    >> reshape(A,2,2)

        ans =

         1 2
         3 4
```

4 Functions and plotting

A MATLAB function is a subroutine that take a list of inputs, performs some computations, and returns a list of outputs. The most common way of defining a function is to use an `.m` file. It should contain:

- The function header (first line)
- A comment that describes what the function does (output of help)
- The body of the function

The function should be stored in a filename with the same name of the function and the `.m` extension. E.g., a file `functionExample.m` can look like this:

```
function [output1,output2] = functionExample(input1, input2)
% This is an example of a function.
% The inputs are switched around
output1 = input2;
output2 = input1;
```

If a function has an output variable defined, its value must be set before the function terminates. To terminate execution of a function prematurely, use the command `return`. Note that, as usual, functions have **local scope**. That means they cannot see any variables defined outside the scope of the function besides their input variables. This means they cannot see the Workspace variables. **If your function needs to read the value of a variable in your Workspace, it needs to be passed that variable as an input.**

Note that the function name defined on the first line of the file can be different than the file name (but this is not a good idea). **In this case MATLAB calls the function by its file name.** To call a function you have defined:

```
>> functionExample(1,2)

ans =

     2
```

Note that by default MATLAB only returns the first output variable. To store all the output variables, define new variables to store them in:

```
>> [a,b] = functionExample(1,2)

a =

     2

b =

     1
```

Anonymous functions allow you to define short, simple functions without needing to create a separate file. The syntax is:

```
>> f = @(x,y) x .* sin(x+y)
```

Here `f` is a **function handle** (a variable that can be accessed to evaluate the function, or to pass the function to another function). The syntax `@(input1,input2)` defines the list of input variables. The rest of the line is the body of the function, and it evaluates the expression and returns its values as the only output variable. **Anonymous functions can access variables in the workspace but their values**

are fixed at the time of creation of the function.

It is a good idea to write MATLAB functions that allow all parameters to be vectors instead of just scalars (if possible). This means that instead of defining a function like

```
>> f = @(x) sin(x)^2;
```

you should define it as

```
>> f = @(x) sin(x).^2;
```

because if `x` is a vector, then the first function definition does not work as exponentiation `^` is not defined for nonsquare matrices, but elementwise exponentiation `.^` is defined for matrices of any size.

Once you have defined some functions of your own, you may want to plot their graphs. All plots are done in **Figures**. A figure is a MATLAB window that contains graphic displays (usually data plots) and GUI components. You can create a new figure explicitly with the `figure` function, and implicitly whenever you plot graphics and no figure is active. By default, figure windows are resizable and include pull-down menus and toolbars. A plot is any graphic display you can create within a figure window. Plots can display tabular data, geometric objects, surface and image objects, and annotations such as titles, legends, and colorbars.

Figures can contain any number of plots. Each plot is created within a 2-D or a 3-D data space called an axes. You can explicitly create axes with the axes or subplot functions. A graph is a plot of data within a 2-D or 3-D axes. Most plots made with MATLAB functions and GUIs are therefore graphs. When you graph a one-dimensional variable (e.g., `rand(100,1)`), the indices of the data vector (in this case 1:100) become assigned as `x` values, and plots the data vector as `y` values. Some types of graphs can display more than one variable at a time, others cannot. Example:

```
>> figure(1)
>> x = -pi:0.01:pi;
>> y = sin(x);
>> plot(x,y)
>> legend('y=sin(x)', 'location', 'best')
>> xlabel('X axis'); ylabel('Y axis')
```

Often it is desirable to use a logarithmic scale on one or both axes. The commands `semilogy`, `semilogy`, and `loglog` work exactly like `plot`, except they apply a logarithmic scale on the `x`-axis, `y`-axis, or both. To learn many other useful tricks on formatting your plots to be more informative, use `help plot`.

5 Relational and logical operators, flow control

The standard relational and logical operators in MATLAB are:

`>`	`greater than`
`<`	`less than`
`>=`	`greater than or equal to`
`<=`	`less than or equal to`
`==`	`equal to`
`~=`	`not equal to`

Relational operators can also be applied to matrices. For example, take two vectors `x` and `y`. To find out, which elements of `x =` are larger than the corresponding elements of, use

```
>> x = [3 2 0 0 5];
>> y = [2 2 2 2 2];
>> x > y
```

```
ans =

     1     0     0     0     1
```

This means that the first and the last element of `x` are larger, the rest are smaller than or equal to the elements of `y`. Each relational operator returns a boolean value indicating whether the comparison is true or false. To perform logical operations on scalar values, use

`&&`	`and`
`||`	`or`
`~`	`not`
`xor(x,y)`	`exclusive or`

Once we have defined logical operations, we can use them to control the execution of the program. The most common one is the `if-else` structure

```
if (a < b && (sin(x) > 0))
    disp('expression is true')
else
    disp('expression is false')
end
```

Note that `end` is MATLAB's keyword for closing flow control statements (equivalent to }) in C/Java-like languages). To exit a flow control block before reaching `end`, use the `break` keyword. Note that calling `break` outside of a control block will terminate execution of the program.

Other flow control statements in MATLAB include the `switch`-statement

```
switch quiz
    case { 10 , 9 }
          grade = 'A';
    case 8
          grade = 'B';
    case 7
          grade = 'C';
    case 6
          grade = 'D';
    otherwise
          grade = 'F';
end
```

used to choose the line to execute depending on the value of the variable `quiz`, the `for`-loop

```
initial_i_value = 1;
final_i_value = 100;
for i = initial_i_value : final_i_value
  Operation_1
  Operation_2
  Operation_3
end
```

used to iterate a fixed number of times while incrementing (or decrementing) the index variable `i`, and the `while`-loop

```
money = 1000;
while (money <= 6000)
  money = money + (1 + (10-1) .* rand(1,1));
end
```

used to iterate a variable number of times depending on some condition that changes over the course of the iteration. They work similarly to other programming languages.

Flow control statements can also be nested, though this is considered bad programming practice and should be avoided if possible. In order to make your code more readable when using nested `if/for/while`-statements with multiple `end` statements, use the smart indentation tool by pressing `Ctrl-I` with the Editor active. When writing long programs, the Editor also allows you to close some flow control blocks by clicking on the `+/-` box next to the `if`-statement.

6 Errors and debugging

This section uses keyboard shortcuts. If your MATLAB is configured to use Unix shortcuts, some things may not work as explained. In this case, change the keyboard shortcuts to Windows mode by entering the menu `File -> Preferences -> Keyboard` and changing both Command Window key bindings and Editor/Debugger key bindings to Windows.

Every once in a while you end up with a program that doesn't seem to run correctly, or sometimes even fails to terminate. In this case you can stop execution by clicking inside the Command Window to make it active and pressing `Ctrl-C`. To investigate where the program goes wrong, it is helpful to use the debugger. Open the program in the Editor and click on a line of the code before the potential problem occurs. Then press `F12` to set a breakpoint on that line, and press `F5` to start execution of the program. Once execution reaches the breakpoint, the program stops and waits for input. You can now:

- Hover over variable names to inspect them (useful for checking that variables have proper values).
- Press `F10` to step through the code line-by-line to see how the execution proceeds.
- Press `F11` to step inside a subroutine to inspect its execution.
- Press `F5` again to continue execution until the next breakpoint or the end.

Note that during all this you can issue commands in the Command Window affecting the execution of the program in run time. Once you have found out where the program is incorrect, press `Shift-F5` to stop execution, fix the offending line, save your `.m` file, and try again. Learning to use the MATLAB debugger is key to writing complex programs that actually work.